With C++ 23 updates

C++ Cookbook

How to write great code with the latest C++ releases

Wayne Murphy

bpb

C++
Cookbook

How to write great code with the latest C++ releases

Wayne Murphy

www.bpbonline.com

First Edition 2024

Copyright © BPB Publications, India

ISBN: 978-93-55515-377

All Rights No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author’s and publisher’s knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

To View Complete
BPB Publications Catalogue
Scan the QR Code:

www.bpbonline.com

Dedicated to

My family and friends
who have supported me,
during this book, and for everything else.

and

My managers at work that believed in me
ever since I first started working in software,
and those along the way.
Thanks et al.

About the Author

Wayne Murphy grew up in Brampton, Ontario, Canada. Graduating from Sheridan College in town. He started work at his first programming job in 1987, and over the years been at many companies, in several different roles, with various technologies. For most of the past 20 years, Wayne has been consulting in his company, Great Leap Forwards Inc. When Wayne is not at work, he is working on some of his own code, and spending time with his family.

About the Reviewers

Maxim Chetruşca is a Software Engineer with almost a decade of professional experience in C++. Throughout his career, he focused on real-time, latency sensitive systems. During his work, he used C++ for the backend of the busiest websites, distributed databases as well as squeezing microseconds from high frequency trading systems. A firm believer of C++ philosophy, he had the chance to use the language versions ranging from 03 to 20 in production, running the code on 4 different platforms. Outside of working hours he spends time learning about new car models with his 2-year-old son.

Kris Jusiak is a passionate Software Engineer with experience across various industries, including telecommunications, gaming, and most recently, finance. He specializes in modern C++ development, with a keen focus on performance and quality. Kris is also an active conference speaker and open-source enthusiast, having created multiple open-source libraries.

Chetan Sachdeva is a Tech Enthusiast, with over 13 years of expertise in C++ and Python in software development. With a dynamic career spanning Automotive, AR/VR, IoT, Android native development, Typography, and Printing RIP. His true passion lies in solving complex problems with data structures and algorithms. His entrepreneurial spirit has led to the successful design of software and products for startups.

Beyond his professional endeavors, Chetan is an avid reader of non-fiction and IT-related books, staying at the forefront of industry trends.

Acknowledgement

I remember that warm summer day in the mid-1970s, when my father’s father told me You should get into and me not knowing what one was. I remember my dad taking me into his work, because they just got their first computer there. I later recall when my parents got us our first family computer, I thought to myself, I will never be bored I have to thank them for that, because I have not been bored yet.

I am very grateful to all those publishing material (books, web-pages, videos) in the goal of wanting people to learn what they know. I hope I have done your efforts justice.

My gratitude also goes to the team at BPB Publications, for giving me a chance to do this.

Preface

C++ Cookbook walks you through all the recent new features. In a cookbook-type style that talks about each new class or function, it shows you in simple terms how to use it. Authored by a software professional, with over 3 decades of experience, their passion for coding shows in the book. This book takes the reader on a tour of what they will need to know to be up to date with the latest C++ abilities.

The book outlines new features, with lots of code examples. They can use it as a reference guide, or progress through each how-to recipe to maximize their knowledge. Sometimes it will give suggestions on the best approach, but mainly wants to inform you of options, and lets you take the right path for your individual situation. The book should help those that have not kept up to date with recent C++ releases. Whether your company was sticking to an older standard, or you are starting with a new product. There are a lot of great new features available, and this book will help you working with the latest and greatest functionality. You can walk through all the chapters with their code examples, or use it as a quick-reference guide for something specific.

After reading it, you will be up to date and will make you and your project work better.

Chapter 1: Working with Concepts – It defines what a concept is, how to use one, and how to create your own. It discusses the different styles how

concepts are defined; whether creating a single constraint on a variable or a function, making a requires clause, or combining concepts to make a conjunction or compound requirement.

Chapter 2: Using the New Core Language Concepts – The chapter initiates a long journey to learn what existing concepts are included in C++. There are concepts for comparison, assignment, checking on the type, and hierarchy.

Chapter 3: Using the New Comparison Concepts – The chapter continues delving into predefined concepts. We look at those relating to equality, and comparison.

Chapter 4: Using the New Iterator Concepts – Our next step into concepts looks at all of the concepts defined for iterators, and also talks about the progression of functionality with types of iterators. We also touch on some other concepts relating to moving or copying values.

Chapter 5: Using the New Object Concepts – We pick up where we left off, talking about concepts relating to the constraints of moving or copying values. We also discuss the similarities and differences between the concepts semiregular and regular.

Chapter 6: Using the New Callable Concepts – We will talk about concepts relating functions, or other mechanisms that we can call to execute some code.

Chapter 7: Const Related Specifiers – With this chapter, we end our lengthy talk about concepts and discuss specifiers relating to defining

what is constant. We go over previously existing specifiers, and talk about newer ones that will help your code be more efficient.

Chapter 8: Concurrent Processing – The chapter discusses threads, callbacks, and different ways for your code to be executed safely by different threads at the same time.

Chapter 9: Coroutines – The chapter presents a new mechanism for executing code. We get into the details of the parts of a coroutine, how to create them, and how to use them.

Chapter 10: Organizing Your Code with Modules – The chapter covers a great new addition to the language. Modules will help you organize your code easier. We get into the details of its components and go through how to using other modules, and learn how to write our own code.

Chapter 11: Introduction to Ranges and Views – The chapter starts discussion about what ranges, views, and spans are. The differences between them, and how to use them are discussed. Since ranges relate to concepts, we will talk about some of the existing range concepts.

Chapter 12: Range Access and Non-Modifying Sequence Functions for Ranges – The chapter continues to discuss ranges, and some of the functionality relating to subranges, sizes, iterating, and comparing.

Chapter 13: Range Algorithms: Sort, Search and More – The next leg of our journey talks about ranges deals with searching and sorting. We can perform functions on a range to get a value, or even a new range by doing a permutation.

Chapter 14: Range Algorithms: Memory and Modification Functions – We focus on memory; moving values around, and doing transformations.

Chapter 15: Views and Range Adaptors – The chapter talks about the many functions that exist so we can get values in a view by calling a range with an adaptor.

Chapter 16: Range Factories and Utilities – We conclude our talk about ranges by showing some exiting new functionality. We can create an infinite view of values, whether by using a simple function or a generator. There is code showing how to even format your range for output.

Chapter 17: New Features for Containers – The chapter covers a lot of ground, dealing with new container types, and simple new functions.

Chapter 18: Making it Easier to Code – There are a number of new features discussed that you will be happy to see such as easier output, new enhancements for strings and ranges, plus an easier way to use enums and bits. There is also new types for dates, times and time zones.

Chapter 19: Making Your Code Cleaner – The chapter covers many recent features for lambdas, new suffixes, optional and expected arguments, plus new preprocessor directives for cleaner code.

Chapter 20: Making Your Code Safer – The chapter touches on some important new features to make code less fragile.

Chapter 21: Making Your Code Faster and Easier to Debug – We conclude by talking about ways to make your code run faster; including some new functions, and new attributes. We also talk about new types that could help you debug your code.

Code Bundle and Coloured Images

Please follow the link to download the

Code Bundle and the Coloured Images of the book:

https://rebrand.ly/de3g86d

The code bundle for the book is also hosted on GitHub at In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at :

business@bpbonline.com for more details.

At you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at business@bpbonline.com with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a

general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Table of Contents

1. Working with Concepts

Introduction

Structure

Objectives

Recipe 1.1: Creating your first concept

Recipe 1.2: Using a concept

Suggestion

Recipe 1.3: Creating a constraint

Recipe 1.4: Creating a conjunction

Suggestion

Recipe 1.5: Making a requires clause

Recipe 1.6: Constraining a function

Recipe 1.7: Creating a requires expression

Recipe 1.8: Multiple requires expressions

Recipe 1.9: Nested requirements

Recipe 1.10: Function templates

Recipe 1.11: Compound requirements

Recipe 1.12: Different types of expressions

Recipe 1.13: Specializations

Conclusion

Points to remember

References

2. Using the New Core Language Concepts

Introduction

Structure

Objectives

Recipe 2.1: same_as

Recipe 2.2: convertible_to

Recipe 2.3: derived_from

Recipe 2.4: integral, signed_integral, unsigned_integral

Recipe 2.5: floating_point

Suggestion

Recipe 2.6: assignable_from

Suggestion

Recipe 2.7: swappable

Recipe 2.8: destructible

Recipe 2.9: constructible_from

Recipe 2.10: default_initializable

Recipe 2.11: move_constructible, copy_constructible

Recipe 2.12: common_with and common_reference_with

Conclusion

Points to remember

References

3. Using the New Comparison Concepts

Introduction

Structure

Objectives

Recipe 3.1: equality_comparable

Recipe 3.2: equality_comparable_with

Recipe 3.3: totally_ordered

Suggestion

Recipe 3.4: totally_ordered_with

Recipe 3.5: three_way_comparable

Suggestion

Recipe 3.6: three_way_comparable with custom types

Suggestion

Recipe 3.7: three_way_comparable_with

Conclusion

Points to remember

References

4. Using the New Iterator Concepts

Introduction

Structure

Objectives

Recipe 4.1: weakly_incrementable

Recipe 4.2: input_or_output_iterator

Recipe 4.3: sentinel_for

Recipe 4.4: sized_sentinel_for

Recipe 4.5: indirectly_readable

Recipe 4.6: indirectly_writable

Recipe 4.7: incrementable

Recipe 4.8: input_iterator

Recipe 4.9: output_iterator

Recipe 4.10: forward_iterator

Recipe 4.11: bidirectional_iterator

Recipe 4.12: random_access_iterator

Recipe 4.13: contiguous_iterator

Recipe 4.14: indirectly_movable, and indirectly_movable_storable

Recipe 4.15: indirectly_copyable, and indirectly_copyable_storable

Recipe 4.16: indirectly_swappable

Recipe 4.17: indirectly_comparable

Recipe 4.18: permutable

Recipe 4.19: mergeable

Conclusion

Points to remember

References

5. Using the New Object Concepts

Introduction

Structure

Objectives

Recipe 5.1: movable

Suggestion

Recipe 5.2: copyable

Recipe 5.3: semiregular

Recipe 5.4: regular

Conclusion

Points to remember

References

6. Using the New Callable Concepts

Introduction

Structure

Objectives

Recipe 6.1: invocable

Suggestion

Recipe 6.2: regular_invocable

Recipe 6.3: predicate

Recipe 6.4: relation and equivalence_relation

Recipe 6.5: strict_weak_order

Recipe 6.6: indirectly_unary_invocable, and indirectly_regular_unary_invocable

Recipe 6.7: indirect_binary_predicate

Recipe 6.8: indirect_equivalence_relation

Recipe 6.9: indirect_strict_weak_order

Conclusion

Points to remember

References

7. Const Related Specifiers

Introduction

Structure

Objectives

Recipe 7.1: Looking at the constexpr enhancements

Recipe 7.2: constexpr for transient allocation

Recipe 7.3: consteval

Recipe 7.4: constinit

Recipe 7.5: constexpr if

Recipe 7.6: if consteval

Recipe 7.7: Some hidden problems with constexpr

Conclusion

Points to remember

References

8. Concurrent Processing

Introduction

Structure

Objectives

Recipe 8.1: std::jthread is the new thread

Recipe 8.2: std::stop_token

Recipe 8.3: std::stop_callback

Recipe 8.4: std::stop_source

Recipe 8.5: std::counting_semaphore, and std::binary_semaphore

Recipe 8.6: std::atomic, and std::atomic

Recipe 8.7: std::atomic::wait, and std::atomic::notify*

Recipe 8.8: std::latch

Recipe 8.9: std::barrier

Recipe 8.10: std::atomic_ref

Conclusion

Points to remember

References

9. Coroutines

Introduction

Structure

Objectives

Recipe 9.1: What is a coroutine

Recipe 9.2: Components of a coroutine: The promise

Recipe 9.3: Components of a coroutine: The state, and the handle

Recipe 9.4: Coroutine restrictions

Recipe 9.5: Generators

Conclusion

Points to remember

References

10. Organizing Your Code with Modules

Introduction

Structure

Objectives

Recipe 10.1: What is a module

Recipe 10.2: What the compiler does with a module

Recipe 10.3: How to use a module

Recipe 10.4: Visibility and reachability

Recipe 10.5: Layout of a module

Recipe 10.6: Cyclical modules

Recipe 10.7: Importing the standard modules.

Recipe 10.8: What are module partitions

Conclusion

Points to remember

References

11. Introduction to Ranges and Views

Introduction

Structure

Objectives

Recipe 11.1: What is a range

Recipe 11.2: What is a view

Recipe 11.3: What is a span

Recipe 11.4: Range concepts

Recipe 11.5: Range primitives

Recipe 11.6: What is a subrange

Recipe 11.7: What is ranges::dangling

Recipe 11.8: What is a view_interface

Recipe 11.9: What is an owning_view

Conclusion

Points to remember

References

12. Range Access and Non-Modifying Sequence Functions for Ranges

Introduction

Structure

Objectives

Recipe 12.1: Review the basic range access functions

Recipe 12.2: The for_each functions

Recipe 12.3: Looking at count, any_of, all_of, and none_of functions

Recipe 12.4: Looking at comparison functions

Recipe 12.5: Search functionality

Conclusion

Points to remember

References

13. Range Algorithms: Sort, Search and More

Introduction

Structure

Objectives

Recipe 13.1: Minimum and maximum functions

Recipe 13.2: Sorting and partitioning functions

Recipe 13.3: Binary search functions, and set functions

Recipe 13.4: Permutation functions

Recipe 13.5: Fold functions

Conclusion

Points to remember

References

14. Range Algorithms: Memory and Modification Functions

Introduction

Structure

Objectives

Recipe 14.1: Heap functions

Recipe 14.2: Uninitialized memory functions

Recipe 14.3: Modifying sequence functions

Conclusion

Points to remember

References

15. Views and Range Adaptors

Introduction

Structure

Objectives

Recipe 15.1: Progressing from views and ranges, to range adaptors

Recipe 15.2: Some simple range adaptors

Recipe 15.3: Using take() and drop() adaptors

Recipe 15.4: Using range adaptors with composition

Recipe 15.5: Using filter() and transform() adaptors

Recipe 15.6: Working with the join() and split() adaptors

Recipe 15.7: Working with the keys() and values() adaptors

Recipe 15.8: Using chunk() and slide() adaptors

Recipe 15.9: Using zip() adaptor

Recipe 15.10: A quick look at reverse()

Recipe 15.11: Looking at special adaptors

Conclusion

Points to remember

References

16. Range Factories and Utilities

Introduction

Structure

Objectives

Recipe 16.1: Factory overview, and the ranges::iota_view factory

Recipe 16.2: The views::repeat, and views::cartesian_product

Recipe 16.3: Creating your range factory

Recipe 16.4: Formatting ranges

Recipe 16.5: Swapping ranges

Recipe 16.6: The utility ranges::to

Conclusion

Points to remember

References

17. New Features for Containers

Introduction

Structure

Objectives

Recipe 17.1: Contains

Recipe 17.2: Looking at std::span

Recipe 17.3: std::counted_iterator

Recipe 17.4: std::is_bounded_array and std::is_unbounded_array

Recipe 17.5: std::to_array converts to std:array

Recipe 17.6: std::erase and std::erase_if

Recipe 17.7: std::flat_map and std::flat_set

Recipe 17.8: Iterators pair constructors for stack and queue

Recipe 17.9: Allow default arguments for pair's forwarding constructors

Recipe 17.10: The push_range() function for queue, stack, and priority_queue

Recipe 17.11: What is std::mdspan

Conclusion

Points to remember

References

18. Making it Easier to Code

Introduction

Structure

Objectives

Recipe 18.1: String formatting

Recipe 18.2: std::print

Recipe 18.3: starts_with and ends_with for strings

Recipe 18.4: Other enhancements for strings and ranges

Recipe 18.5: using enum reduces typing for enums

Recipe 18.6: New date features

Recipe 18.7: New time features

Recipe 18.8: Timezone library

Recipe 18.9: std::midpoint

Recipe 18.10: includes pi and e

Recipe 18.11: Bit manipulation

Recipe 18.12: Designated initializer for aggregates

Conclusion

Points to remember

References

19. Making Your Code Cleaner

Introduction

Structure

Objectives

Recipe 19.1: Familiar template syntax Lambdas

Recipe 19.2: Lambda parameter packs

Recipe 19.3: Literal class types in non-type template parameters

Recipe 19.4: Multidimensional subscript operator

Recipe 19.5: std::make_shared supports arrays

Recipe 19.6: CTAD improvements

Recipe 19.7: Deducing this

Recipe 19.8: Transforming auto

Recipe 19.9: char8_t and std::u8string types

Recipe 19.10: Suffix for std::size_t

Recipe 19.11: Named universal character escapes

Recipe 19.12: Monadic operations for std::optional

Recipe 19.13: Using std::expected

Recipe 19.14: Pre-processing directives elifdef and elifndef

Recipe 19.15: invoke_r

Recipe 19.16:

Recipe 19.17: Bit initialization

Recipe 19.18: Extend init-statement to allow alias-declaration

Conclusion

Points to remember

References

20. Making Your Code Safer

Introduction

Structure

Objectives

Recipe 20.1: Fixed width floating-point types

Recipe 20.2: Conditionally explicit constructor

Recipe 20.3: #warning

Recipe 20.4: std::start_lifetime_as

Recipe 20.5: std::out_ptr(), std::inout_ptr()

Recipe 20.6: Overloads of std::to_chars() and std::from_chars()

Recipe 20.7: constexpr std::bitset

Recipe 20.8: std::to_underlying

Recipe 20.9: Direct initialization

Conclusion

Points to remember

References

21. Making Your Code Faster and Easier to Debug

Introduction

Structure

Objectives

Recipe 21.1: [[likely]] and [[unlikely]]

Recipe 21.2: [[assume()]]

Recipe 21.3: std::unreachable

Recipe 21.4: string::substr is faster

Recipe 21.5: string::resize_and_overwrite

Recipe 21.6: PMR containers

Recipe 21.7: Comparing integrals

Recipe 21.8: std::endl vs "\n"

Recipe 21.9: Synchronized stream output

Recipe 21.10: std::source_location

Recipe 21.11: std:stacktrace

Conclusion

Points to remember

References

Index

Welcome to C++

C++ is a wonderful language. You can often be working with high-level abstractions, but still be able to dig down deep and work with raw memory. C++ has been around for decades, and with the efforts of many good people out there, is still evolving. We can assume that this means that it will probably be around for decades more. It is on lists of most used languages (sometimes in the top 10, and if not, then certainly the top 20), as well C++ is on lists of programmers' most liked languages. C++ is not owned by a big company, has multiple companies making compilers for it, and many companies making tools and libraries for it. It runs on different operating systems, for purposes big and small. The governing body meets regularly to discuss, debate, and decide what will go into the next release. Releases were decided to be every three years, which they have done in the age of modern C++ (11, 14, 17, 20, 23, and they are already talking about C++26). To be precise, the releases are defined every three years. At the time of writing this book, at the end of December, C++23 was still being voted on whether it is officially released.

We could not talk about everything to do with C++ just in one book, and so we are here to talk about the latest couple of releases; C++20 and C++23.

Recipe 1: Who is this book for?

We want to know who should read this book.

It can appeal to different types of audiences.

Of course, we would like to say anybody and everybody can read this book. However, this is a cookbook focusing on the newest releases. If you are just starting out knowing little about C++, then this probably is not the best first book for you. Having said that though, today's world lets us learn in different media in different ways. If you have a handful of websites and other learning video channels that teach you the basics, then yes, you can buy the book as a secondary source of learning.

This book does not only discuss the basics but focuses on new features. In that regard, it can be used by any who knows the basics but is not kept up to date with all the latest features. If you look at the following figure, we would say that we are trying to focus on what you will want to know to get caught up on the latest features.

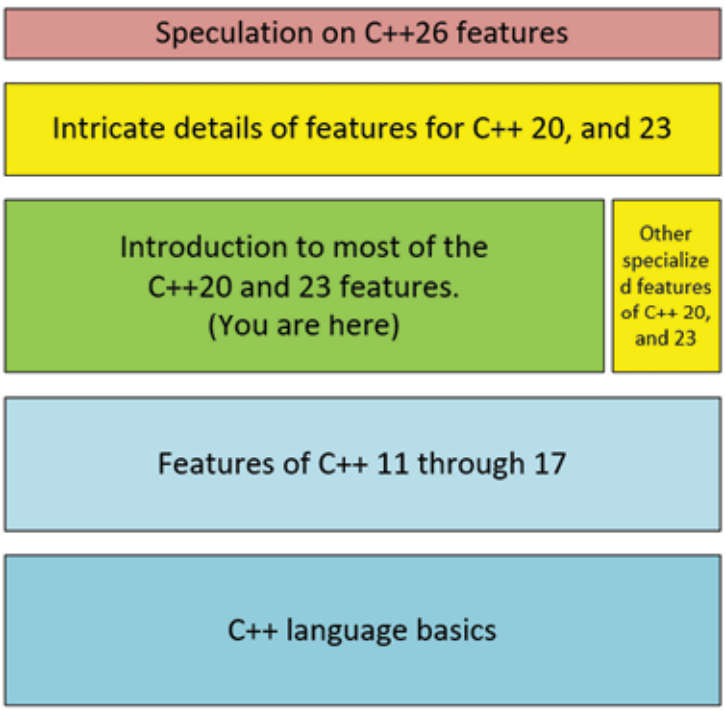
Speculation on C++26 features
Intricate details of features for C++ 20, and 23
Introduction to most of the C++20 and 23 features. (You are here)
Other specialized features of C++ 20, and 23
Features of C++ 11 through 17
C++ language basics

Figure 1: What this book is about

Recipe 2: What is the format for each chapter?

How is each chapter planned out?

It is a cookbook-type design, with many problem and solution topics for various Recipes.

Each chapter has a particular focus:

We have an or overview of what the chapter is about.

The Structure is given, which lists the features or topics that we will talk about in detail.

We have a brief statement on the Objectives of the chapter.

Then, we iterate over our which are individual topics in the chapter. Sometimes, a recipe can be about an idea, but often it is about a particular class or function. There can be between 5-15 recipes in a chapter. We have tried to provide lots of code samples to help illustrate what the recipe is about.

Next, we give a which summarizes what you have learned in the chapter.

We give reminders of some important facets in the Points to remember section.

Lastly, we provide aside from needing to say where the information came from, which is also a huge and sincere thank-you to those who provided

information and helped inform others. There are probably more resources that we could have mentioned. For brevity's sake, if we saw similar information on a handful of different websites and videos, then we jotted down what we thought best presented the information.

Recipe 3: Is C++23 worth reading about?

If there have been so many releases, can we just learn it as we go?

You probably can learn a bit as you go, however the good and bad part about C++, is that it has evolved over the decades. So yes, you could continue using nothing but new and delete, and maybe you could get by and use a few new classes as you trip over them. The result though, is that the code, and coder, will not be as good as they could be. Some things get deprecated, but a lot of code is still around due to historical use. Everybody says you should not use function but now use function And in truth, we are probably on function D() in some cases. You should want to at least be aware of the latest features. You may not want to, or need to, rewrite your entire code-base because new features exist, but for new functionality, you should understand what is in the latest release(s), to make an informed decision on when to use them. Compilers are smarter, computers are faster, learning tools are better, and so using newer features will not make your job more difficult. Releases are created based on input like yours, to make coding with new features safer, smarter, faster, and easier.

To get into slightly more detail, the C++20 release was huge. It was arguably the biggest C++ update ever in terms of features. What is more, it helped to define a direction on where the language should head. C++20's Big Four features will influence other features for years to come. C++23 is not as big as C++20, partially, because of the global pandemic, and admitted partially because there were multiple changes from C++20. Having said that, C++23 is

an impressive release on its own merit, which will change how coders create software.

Refer to the following figure:

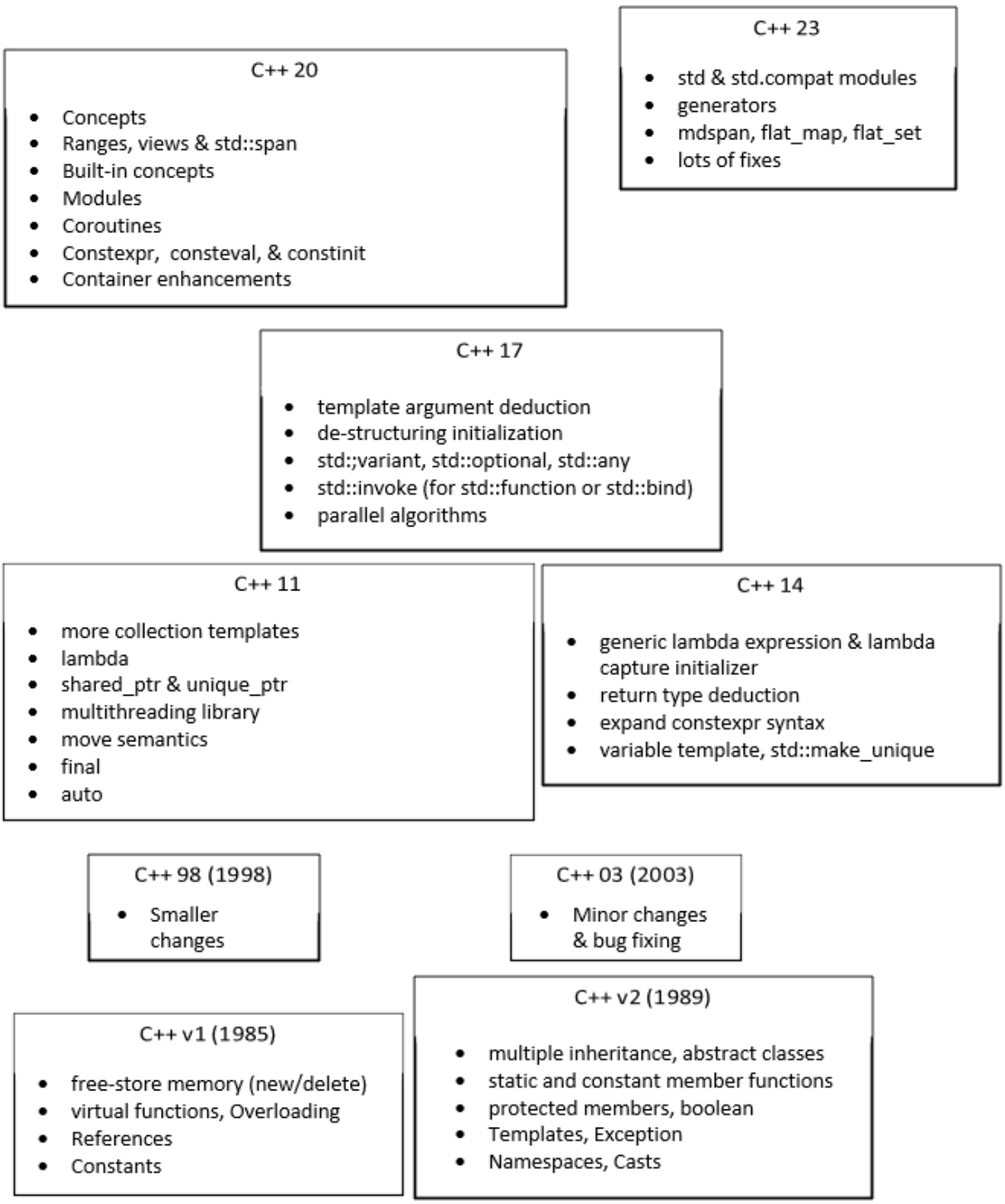

Figure 2: Overview of C++ releases

Recipe 4: Is the book just about C++23?

We want to learn more about the latest version, but have not focused too much on other recent releases.

This book talks wants you to learn the latest, but understands if you might have missed something along the way.

The book has tried to include everything that has been defined in the C++23 release. A couple of features might be vague or short, partially because some features are more important than others, but also a couple of C++23 features are still a bit vague. They are included in the release, but no one has implemented the feature, and there is not much information about them outside of the spec. As we have said, C++20 was such an important release, that some parts of C++23 are tough to understand if you do not know about the related C++20 features. We do talk quite a bit about the C++20 release, and even some parts of the C++17, but that is not to distract you from learning about C++23, but here to help you understand how we got to C++23.

Recipe 5: How to work with the code

How do you use the code samples provided?

We can walk you through some of that.

Most of the code samples were built and run in Visual Studio 2022. We have used Visual Studio Code as well. There are two main steps to use C++23 in Visual Studio. The first is to run the Visual Studio Installer program and enable using the Latest C++ tools. Refer to the following figure:

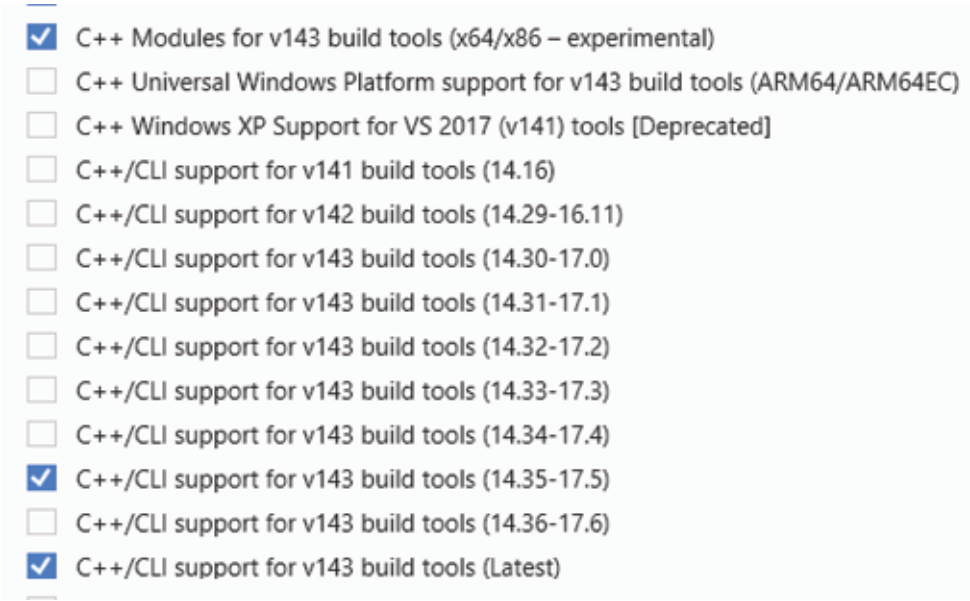
C++ Modules for v143 build tools (x64/x86 – experimental)
C++ Universal Windows Platform support for v143 build tools (ARM64/ARM64EC)
C++ Windows XP Support for VS 2017 (v141) tools [Deprecated]
C++/CLI support for v141 build tools (14.16)
C++/CLI support for v142 build tools (14.29-16.11)
C++/CLI support for v143 build tools (14.30-17.0)
C++/CLI support for v143 build tools (14.31-17.1)
C++/CLI support for v143 build tools (14.32-17.2)
C++/CLI support for v143 build tools (14.33-17.3)
C++/CLI support for v143 build tools (14.34-17.4)
C++/CLI support for v143 build tools (14.35-17.5)
C++/CLI support for v143 build tools (14.36-17.6)
C++/CLI support for v143 build tools (Latest)

Figure 3: Showing we are using the "Latest" build tools, in the Visual Studio Installer

The next is to go into your project settings and select if you want to use the Preview release of the C++ Standard. This would equate to having /std:c++latest in your makefile. Refer to the following figure:

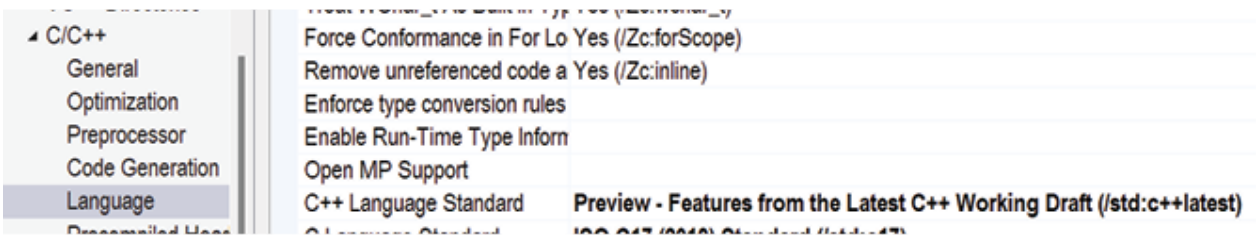

Figure 4: Visual Studio project setting to use the latest C++ standard

Once you have done those two tasks, then you are all ready to code many but not all of the latest features in the release. That is because they have not stopped implementing all of the new features in the compiler. If there is something you want to check, you can go to the release pages: https://en.cppreference.com/w/cpp/20#Compiler_support and https://en.cppreference.com/w/cpp/23#Compiler_support to see if your compiler has implemented that feature yet. Once the C++23 release becomes official, you should not have to opt for the experimental version of the compiler, but we wanted to show how we got the code running.

If you are more familiar with clang, or gcc, then you still have the opportunity to use those compilers. They too, are powerful tools, and you have more options to use a user interface such as Visual Studio Code, or some other integrated graphical editor and builder. You could also just edit the files in your favorite editor, and use the command line to build them. Of course, there is also the choice of using different operating systems to run the samples on. You probably have your environment already set up, but as a quick reminder you could refer to the Clang installer at and the gcc installer can be found

A side note, changes for a release are divided into C++ core language changes and C++ library feature changes. Changes are made by submitting a paper documenting your change, and normally after several revisions, it is decided whether or not to include the feature in the release. The official site is but they state it is more of a legal treaty and not something you can easily learn from. Whether compiler manufacturers have implemented the feature, does not directly determine whether it will eventually be included in a release. Every paper is assigned a number, and every revision on the paper changes the suffix. That is why, when you see features listed, you will see the paper referenced, a fictitious example being P1234R12. You can see the full list at

The MSVC compiler is pretty good about the number of features it has implemented, but there are quite a few gaps, compared to the GCC and Clang compilers, and they have quite a few gaps as well. After those main three, you will have to be careful about what you want to look at. There are several other compilers, but some of those have big gaps stating they have implemented C++20 features.

The other thing to try is to see if the C++ feature is available in the compiler, and possibly the list has not been updated. Most new features have an associated macro associated with them, and you can use an #ifdef to see if the feature is available. You can do that by looking at the full list at https://en.cppreference.com/w/cpp/utility/feature_test , but it seems easier to look on the individual class/feature page and it will list the associated macro there, such as https://en.cppreference.com/w/cpp/utility/format#Notes.

You can also look at the Visual Studio release notes https://learn.microsoft.com/en-us/visualstudio/releases/2022/release-notes to see what new features and fixes they have worked on.

For GCC, you can enable C++23 support by following the instructions here The main thing to do is to have the standard setting be “-std=gnu++2b”. On this page, it also lists what C++23 features are available in GCC, and what macro to check for.

With Clang, it too uses and you can look at the status here

There are other compilers that have partial support for C++23, and you can find out the list, or the status of the compiler you current use, at

Visual Studio Code is another popular IDE and would act as a UI for the compiler you have installed. You would just need to ensure the installed compiler is up to date, and you have the appropriate setting to tell the compiler what standard to use.

As mentioned, the code samples in our recipes relate to the various chapters and sections in the book. If we are on the second section/recipe in Chapter then you should find the code will look like:

```
  // ...

 }

};
```

If we have multiple iterations of code in the same recipe section, or multiple code examples in the same recipe section, then we will append an extra number showing the iteration:

```
// ...

}
```

```
};
```

Most code uses static functions, just for ease of use. We recommend you do not do that since we just used that format for simplicity. That allowed us to call the functions for that chapter without a lot of instantiations. We could simply call them from a method at the bottom of each code file of each chapter:

```
// ...

}
```

This allows you to try the code with the compiler on your computer, or to paste the chapter's code into an online compiler, and add a simple function to call it, such as:

```
    LookAtChapter20Topics();

}
```

We give much praise to the online compiler https://godbolt.org/ , which we used to verify whether code features were available yet on the clang and gcc compilers.

On your computer, there is a main() function that you can call all the code samples if you wish:

```
{
    LookAtChapter1Topics();
    LookAtChapter2Topics();
    LookAtChapter3Topics();
    LookAtChapter4Topics();
    LookAtChapter5Topics();
    LookAtChapter6Topics();

    LookAtChapter7Topics();
    LookAtChapter8Topics();
    LookAtChapter9Topics();
    LookAtChapter10Topics();
    LookAtChapter11Topics();
```

```
    LookAtChapter12Topics();

    LookAtChapter13Topics();

    LookAtChapter14Topics();

    LookAtChapter15Topics();

    LookAtChapter16Topics();

    LookAtChapter17Topics();

    LookAtChapter18Topics();

    LookAtChapter19Topics();

    LookAtChapter20Topics();

    LookAtChapter21Topics();

}
```

With the header simply defining the chapter functions as follows:

```
#pragma once
```

Recipe 7: Conventions used

How do we know what each font/style that is used means?

You have come to the right place.

If you understand C++, then we probably do not need to walk you through every individual font and style to say what it means. The main style to point out is when we talk about a or a and those are made bold to grab your attention. There are, of course, times when we refer to an external source for information, and then is when we mark them in italics which are later listed in the References section. We have tried to present information through variable means. We talk about a subject, and give you a standalone code sample. Then when it seems helpful, we will show you the output. We provide information in lists, and a few diagrams as well. The main learning source would be the code, where we have tried to provide helpful and short examples.

CppReference.com is an incredibly helpful website, and most likely something that we already have bookmarked and visit regularly. Having said that, some examples are not always that brief and that clear. Hopefully, our examples are simpler and help you understand how you might be able to apply that feature to your code. Often with books, people use a theme with their code examples, which is better than always talking about Widgets, Foos, or Bars. Some resources might focus on a company making bicycles, some do not try to abstract and their code samples get a bit too complicated.

Recipe 8: Do not do as we do, do as we say

Should you strictly follow how we have coded in our examples?

No.

The code samples illustrate and give you an idea about part of what a feature does. We try to keep the examples brief, and aside from that being a good thing, seeing the whole function on a screen, remember that even less fits on a

printed page. There will be many an excuse for short variable names, spacing (vertically and horizontally), plus things that can cause big battles like braces placement. We use a code formatter, which we do not have much control over. We can borrow an expression what happens in a code-example, stays in a Follow your company coding guidelines, and if you do not have any, discuss about making some. Follow coding guidelines that your colleagues and friends follow, for there is something to be said about it being easier to read when things are consistent. We recommend you follow the gist of the code sample, not things such as:

Variable and function names, or casing.

Vertical or horizontal spacing.

Placement of brackets, braces, and other symbols.

Massive underuse of const-related qualifiers. Some are new, all are good, and should be used as much as possible.

Overuse of It is quick and easy to sample code, but best not to litter your code with them.

Lack of modules. They seem like a great idea but we have not used them all the time for the code samples, because we did not want to be using something before you read the chapter about them.

We will reuse the last answer, relating to several other features that we did not use in the first chapter. It is because we did not want to use a feature before we talked about it, and there are a lot of chapters in this book.

If it were not about spacing out the features, and not using them until we talked about them, the next biggest reason was that at the time of writing the code sample, a lot of functionality was not implemented yet.

Overuse of or even any use of it. We used it because it makes it more obvious when we break up a line. If we hid some \n in the middle of text, then you might miss it and wonder what is going on.

We do use auto often for variables, but not always. When we explicitly state the variable type, sometimes it is just to show you what the return type is.

Similarly, we often explicitly state what the types are in a collection. (declaring std::vector numbers instead of using CTAD (declaring std::vector numbers We just wanted to be clear what we were doing in our code

Perhaps an obvious one, but the book is to show you how the code runs. Sometimes, we have output that does not have to exist, but it is there to show the result so you do not have to debug the code to understand what is going on.

You should try to use the latest technology that you have available. We are not asking you to use what is on the bleeding edge of technology. Often a newer release can have some new bugs and issues to deal with, but try to use some reason. This is a segue to discuss why you see std::cout instead of std::print in every chapter, or why not all the samples are not in modules. A simple answer for that is that we did not discuss them in the first chapter, and we did not want to use something before we discussed it.

Recipe 9: Do you need to learn from anywhere else other than this book?

Do you need to learn more about C++ from anywhere else?

Why would you not want to?

There used to be only a few sources from where you can learn C++, such as:

Colleagues

A trip (normally to a bigger city) to visit book stores

There were probably some user groups out there, but it was tough to know they existed.

Occasionally, there would be a conference you could travel to and attend.

Nowadays, you can learn so much about the experts talking about C++ in videos, websites, blogs, podcasts, and more. It is recommended to go see what is out there, what presentation or learning style suits you, and go from there. People learn better when they are comfortable with the teacher, and there are so many people that dedicate their own time to show you interesting things about the language.

On another note, the focus of this book is on the latest couple of releases of the C++ standard library. There are many great, useful libraries out there, but they just were not in the scope of this book. There would not be many who could code an entire project without using any third-party libraries, and they too are another source of learning. You can check out what else is out there for you to solve your coding problems. (see references)

Since it relates to the topic, we cannot thank enough https://en.cppreference.com for all the information they have. Whether it be the status of the compilers for the various features, making the official C++ features a bit more understandable, or all of the same code, it seems to be one of the best sources to learn more about them. We have certainly referenced them in this book for various references, but the number of references does not measure how valuable a resource they are.

Recipe 10: What is next?

Do you need to learn more after you learn about C++23?

Yes, the fun never ends!

As we have mentioned a bit about the process of C++ releases, there are regular meetings on the next standard and release features. As we can see by looking at https://en.cppreference.com/w/cpp/compiler_support/26 the GCC and compilers have a head start on some of the features that have already been decided will be in the C++26 release. This leaves us all looking forward to all the upcoming features.

References

CPPreference. (n.d.). Libs. Retrieved from https://en.cppreference.com/w/cpp/links/libs

CPPReference. (n.d.). CTAD. Retrieved from https://en.cppreference.com/w/cpp/language/class_template_argument_deduction

CHAPTER 1

Working with Concepts

Introduction

Concepts were a big new feature (part of The Big Four features) of C++ 20. The proposal had been around for several years, hoping to become part of the official C++ language, but due to all the implications and scenarios, it took a long time to be fine-tuned to get eventually accepted.

The idea of concepts is about as old as templates (which were first introduced in C++ 11). Unfortunately, we only got to experience templates by themselves for over a decade, which is longer than what people would have wanted. Templates are powerful, but they were found to be too vague in dealing with all the entities that were passed to them, such as:

If you pass a string, when at first when the code was created, you were thinking you would only be passing numbers, or perhaps when passing things such as collections;

Passing complex structures, although you initially assumed you would use just simple types.

Templates were poor at type-checking cleanly and/or easily; the problem being, when you pass something specific into a vague template, you end up with tricky runtime errors, which of course is too late to deal with in a clean manner. At best, you get extremely confusing compiler errors. Compilers (that handled templates properly) tried to do their best to tell you what the problem was, but it took a lot of effort (and patience) to

understand what the compiler was trying to tell you about the coding issue that was related to the template.

Concepts put restrictions on templates, and so, the runtime and even compile time troubles go away if you give a little bit more of a description to the types of data you are passing in.

As we progress, you will learn that this little bit of extra effort required, makes things clearer, a lot easier to use, and a lot more powerful too. Concepts are a breakthrough way of simplifying how you will code going forward. If you worked with templates, then you will greatly appreciate how confusing and frustrating it was, to deal with template compile errors.

The constraints for concepts are checked at compile time, and the error messages are much more understandable than before when using just templates. Let us focus on the new era of concepts because they are huge - you will soon always use concepts, and rarely use templates on their own.

To answer the question that you might have been thinking: If the book includes the latest version of C++ which is C++23, then why start the book with a chapter about C++20?Here are a few reasons:

Just in case the reader has previously used a version pre-C++ 20.

Concepts are such an important new feature that it is worth reviewing.

C++ 20 was a good release in that it had a lot of big features, but there were also many shortcomings and little issues that needed to be fixed/improved in C++ 23. So, without getting into specifics about it, we

can say that C++20 is when concepts were introduced, and C++ 23 is when concepts were ready for wide use. Perhaps, some coders did not rush into C++ 20, thinking it was too new to use.

Concepts are used in other features of C++ 20 and C++ 23, so, it is important to have a solid understanding of them.

Quick takeaways of concepts would include:

Concepts (designated by the new keyword are evaluated at compile time.

They constrain parameters (template arguments).

They help make the code cleaner, and certainly make compiler errors cleaner and easier to understand.

Concepts simplify how templates are used (and reused). You can think of concepts as an interface (as you might in Java or C#), or (almost) as a type, which has defined constraints, or you can think of them as a set of requirements – to which your entities will match up with. At times, they may not appear to add much, but for those that used templates a lot before, they will soon appreciate the new feature.

Structure

In this chapter, we will cover the following topics:

Recipe 1.1: Creating your first concept

Recipe 1.2: Using a concept

Recipe 1.3: Creating a constraint

Recipe 1.4: Creating a conjunction

Recipe 1.5: Making a requires clause

Recipe 1.6: Constraining a function

Recipe 1.7: Creating a requires expression

Recipe 1.8: Multiple requires expression

Recipe 1.9: Nested requirements

Recipe 1.10: Function templates

Recipe 1.11: Compound requirements

Recipe 1.12: Different types of expressions

Recipe 1.13: Specializations

Objectives

By the end of this chapter, the readers will be introduced to the concept, with practical help from the various recipes added. They will also learn how to create a constraint for a template or a function, what the required keyword does and what are the requires clauses. Lastly, the chapter will also discuss the different types of requires expressions.

Recipe 1.1: Creating your first concept

Problem: We want a better way to use templates.

A big new feature in C++ 20 is called concepts.

You will always need to include the concepts header (concepts), which defines concepts and a lot more, that we will get into later.

Our header will define an overly simplistic entity; since we are working with a cookbook of recipes, we want something that counts recipes. There could be a few or many thousands of recipes. The main idea to focus on is that the number is whole, without fractions.

```
// chapter1.h

#include

template T>

concept PageCount = std::is_integral_v;
```

One thing to note is that there are different ways of specific something. For example:

```
template T>
```

```
concept PageCount = std::is_integral::value;
```

Under the covers, the compiler will create slightly different code, but for simplicity's sake we will say the 2 PageCount definitions are the same.

Note: One tricky area about concepts (or about templates, or about several features of C++ in general) is that it can be written in different ways. Most times it will not make a difference how you define it, but do not be alarmed when you see something written differently.

So now, our PageCount has the constraint that T must obey the class template (defined in C++ 17), saying that the type must be an integer. Now, again, this does not give any big obvious gain over just using a template, but using concepts will give you:

Compile time type-checking

Clearer messages, if there is a compile-time problem

As we will see soon enough, we can build off of the concepts we define.

Recipe 1.2: Using a concept

Problem: How to restrict a function argument to only allow types with specific constraints?

Solution: Concepts can be used in different ways. One handy way is a function argument.

To use our concept, we could simply have:

```
#include "chapter1.h"

PageCount auto AddPageCounts(PageCount auto chapter1, PageCount auto chapter2) {

return chapter1 + chapter2;

}
```

This function adds up the number of pages in the first two chapters of our cookbook. If we did not know what PageCount was, we get a hint it is a concept (versus a type) because it includes the keyword auto in the declaration. You might ask why we just added two values instead of a collection of values. Here are a couple of reasons:

We could have defined many parameters in the function, but we wanted to keep it simple for our example.

Since PageCount is a concept and not a type, we could not have coded our parameter as a collection of concept values. To accomplish that, we would have to define another concept. The C++23 compiler does not understand if we had AddPageCounts(PageCount* chapters, int numberOfChapters) or used a contain, such as AddPageCounts(std::vector

Suggestion

A brief aside, and a quick reminder, that you might be asking why even bother with a concept or a template. Why can we not simply say that the method uses an integer?

You can definitely do that, but remember that C++ has lots of variations for the integers of various sizes, whether it is signed or not, as shown in the following list:

Short (or short int, signed short, signed short int)

Unsigned short (or unsigned short int)

Ont (or signed, signed int)

Unsigned (or unsigned int)

Long (or long int, signed long, signed long int)

Unsigned long (or unsigned long int)

Long long (or long long int, signed long long, signed long long int)

Unsigned long long (or unsigned long long int)

When you have so many types of integers, a template and type check can be very handy. If you just use a template, then you are confining all of your variables to be of that specific type. That is fine if you are defining a library function that can be used with different types, as long as each particular invocation of that function uses the same type each time. In other words, if it was just a template, then you could call it with an int and an int. Another time, you could call it with two unsigned longs. The template gives you some flexibility, as long as you are consistent in each usage.

Implementing your function using a concept, allows you to mix your types, as long as the types adhere to the requirements specified. Whether writing the functions or calling the functions, concepts add that extra bit of clarity to help you understand how to use the function. It also gives that extra safety, so you know the function will be used as expected.

We can create a little function that uses our

```
static void AddSimplePageCounts() {

unsigned short int ch1 = 4;

long long ch2 = 123456;

auto chapterTotal = AddPageCounts(ch1, ch2);

}
```

Although we are passing different types of values to our function, since it knows that it adheres to an integer type, it can add the two values together.

Recipe 1.3: Creating a constraint

Problem: How to reuse a concept to create another constraint?

Solution: Using a concept as an argument and returning a Boolean can be rather handy.

You will start to realize the potential of concepts; instead of having a method such as AddPageCounts that returns the same type of value as you provided, you can have a function that can also act as a constraint:

```
// function that uses our previously defined PageCount

bool IsEvenNumberOfPages(PageCount auto n) {

    return (n % 2 == 0);

}
```

As mentioned earlier, things get more interesting when we start to combine concepts and constraints. We can do that, although, using logical operators.

Recipe 1.4: Creating a conjunction

Problem: Is it possible to combine multiple concepts?

Solution: You can use logical operators to leverage existing concepts into new ones.

Let us tweak our existing concept since it does not make sense to have a negative number of pages:

```
// a concept that uses another C++ 17 class template

templateT>

concept UnsignedPageCount = std::is_unsigned::value;

// a new concept combining other concepts

templateT>

concept UnsignedEvenPageCount = UnsignedPageCount &&
IsEvenNumberOfPages;
```

So now, at compile time, you will get a readable error message if you do not assign an even integer to

Note: This definition that contains a combination of multiple concepts is called a conjunction. If instead you use the logical OR (||) instead of the logical AND (&&), it is called a disjunction. Naturally, it satisfies the definition if either of the two constraints are true. You can overload (basically define a sub-type) for more detailed concepts.

Suggestion

In case you are not that familiar with them, you will see that the book has already used a couple of constraints such as std::is_integral_v and These are in the library header which was introduced in C++ 11. A lot has been added in subsequent releases. Although not used every day, it would be a good idea to take a look in there and see how those definitions can be used when you make your own concepts.

Another type of constraint is called which is an expression that can be evaluated at compile time. It cannot be broken down any further, such as the case if it is evaluated to a If we look at some code:

```
template T>

concept LargePageCount = std::is_integral_v && sizeof(T) > 4;

void PrintPageCount(LargePageCount auto count) {

std::cout << "That is a big chapter!: " << count << "\n";

}

void PrintPageCount(PageCount auto count) {

std::cout << "This is a small chapter: " << count << "\n";
```

```
}

static void PrintPageCounts() {

unsigned short int ch1 = 4;

long long ch2 = 123456;

PrintPageCount(ch1);

PrintPageCount(ch2);

}
```

We defined a new concept named This is a conjunction, combining two different constraints. The first is the integral check we have been using, and the second is checking the size. This sizeof check is atomic, since it is either true or false. The output of the function is:

Figure 1.1: Output from PrintPageCounts

Although our two PrintPageCount functions have different signatures, we had to place the one that uses LagePageCount first. Otherwise, when the code ran, it would have used the first function it could, and even with a large value, it would still work with the one that uses the regular PageCount parameter.

Note: Just like the rules of evaluating a math equation, there are rules, or orders of precedence, on how constraints are judged. This is something readers must know.

Another keyword introduced in C++ 20 is The requires keyword is used to specify extra constraints on the arguments or on the declaration itself. You could use requires on normal templates too. Although our focus is on the new features, note that you can always create your constraints.

Recipe 1.5: Making a requires clause

Is there more than one way to define a concept?

The requires clause is another mechanism that allows you to state a constraint.

We will change our function from earlier from:

```
templateT>

concept UnsignedPageCount = std::is_unsigned::value;

// a new concept combining other concepts

templateT>

concept UnsignedEvenPageCount = UnsignedPageCount &&
IsEvenNumberOfPages;
```

We can also have our code look like this, which is a bit different, but we would get a similar result:

```
templateT>
```

```
    requires UnsignedEvenPageCount

bool IsEvenNumberOfPages(T n) {

return (n % 2 == 0);

}
```

In the previous concept example, we used a function to help constrain a type. In a way, this concept requires a type to help constrain a function.

Again, for the most part, it can be used in the same manner as before – however, this version is written with a requires clause after the template parameter list. In some scenarios, you can state the requires constraint after a function declarator.

Recipe 1.6: Constraining a function

Can you use a requires clause to constrain a type or a function?

The requires clause defines the constraint before your function code.

To restrict the type used, we could rewrite a previous example and have our fictitious code as:

```
templateT> requires PageCount

T AddPageCounts(T a, T b) {

return a + b;

}
```

This will ensure that we do not provide something like strings that would append, or collections that get added together. Keep in mind, that this code would not pass scrutiny since it does not do bounds checking.

The preceding code snippet shows an example of the requires clause, where the keyword states the constraint on the template argument or the function declaration.

Recipe 1.7: Creating a requires expression

Do we need to define the constraint in the preceding code, where we use the type?

A requires expression allows you to define your constraints in a code block after the concept name.

The other scenario is to use a requires expression. This type of constraint is similar to being atomic, where it evaluates the expression of type bool to ensure the condition is true:

```
templateT>

concept IsAddable = requires (T a, T b) {

a + b;

};
```

In this scenario, it does not return any numerical value. The constraint just says that it is possible to add two values together. So, in this snippet, you could add two strings together.

Recipe 1.8: Multiple requires expressions

Does stating the requires expression in a code block mean we can state multiple constraints?

The requires expression can hold/define many facets of the concept (whether you call them capabilities or restrictions).

Note: The syntax in the prior example. Within the brackets, it has the expression a + b; only, although when used, the result would be non-Boolean. This constraint is a Boolean saying it is possible to add two values together.

To tighten up that code, if we wanted to ensure it was integers we were trying to add, we can have multiple expressions:

```
templateT>

concept ArePageCountsAddable = requires (T a, T b) {

    a + b;

    std::is_integral_v;

};
```

Understandably the example is rather contrived, needing to check if numbers can be added - when inherently we know they can be. However, it is a simple example of the feature’s capabilities. In the expression, the first expression on line 3 is called a simple and the second on line 4 is called a type

Another usage is that of a compound which defines constraints on return types:

```
templateT>

concept HasDoubleIt = requires (T t) {

    { t.DoubleIt() } -> std::convertible_to;

};
```

For this snippet, the concept HasDoubleIt has the constraint that the type must contain a function named The syntax says the result (indicated by the must be convertible to an Keep in mind that the return constraint may not be a simple type. The return type being convertible means that it can be converted to an which has more leeway than if your constraint was just using

```
    { t.DoubleIt() } -> std::same_as;
```

Recipe 1.9: Nested requirements

If we can have multiple constraints in a requires expression, can we have multiple requires also?

There is the ability to combine multiple requires statements to create a more complex concept.

The last scenario to discuss is nested which relate to constraints on type parameters by reusing previous concepts:

```
templateT>

concept UnsignedOddPageCount = UnsignedPageCount && requires(T) {

   requires !UnsignedEvenPageCount;

};
```

The usage would be as follows:

```
// is fine since we provided an unsigned

UnsignedOddPageCount auto a = 43u;
```

```
// will not compile since we provided an even number

UnsignedOddPageCount auto a = 42u;

// would not compile, since the number 43 is signed

UnsignedOddPageCount auto a = 43;
```

So, remember that there are different ways to write the same (or at least very similar) thing, but the UnsignedOddPageCount example shows that you can reuse a constraint as a requirement. Just be wary that it is dangerous to define a constraint as the negation of another constraint.

Recipe 1.10: Function templates

If we can define concepts in different ways, is there one way that is better than the others?

How you define your concept can depend on your style and what works best in each situation.

If we think of different ways to write the same thing, we can go over function templates, as follows:

```
#include

template T>

concept PageCount = std::is_unsigned::value;

auto AddPageCountsUsingAbbreviatedFunctionTemplate(PageCount auto a, PageCount auto b) {

return a + b;

}

template T> requires PageCount
```

```
auto AddPagesCountsUsingRequiresClause(T a, T b) {

return a + b;

}

template T>

auto AddPageCountsWithTrailingRequiresClause(T a, T b) requires
PageCount {

return a + b;

}

template T>

auto AddPageCountsWithConstrainedTemplate(T a, T b) {

return a + b;



}
```

We have seen our definition of PageCount before. Further, we have also seen our example of although here it returns auto instead of which might

be a bit safer. You might see functions such as which specify the requires and then the function. For it moves the requires to after the function name.

Lastly, as you see with instead of specifying we could embed the concept name into the template and the compiler should generate the same code.

Note: You will have to follow the corporate coding guidelines, or at least try to be consistent.

Recipe 1.11: Compound requirements

What requirements can we place on a template type?

Concepts allow to put various constraints, similar to an interface.

When you look at the next example of compound you realize you could use concepts (with or without almost like an interface:

```
templateT>

concept DoesMagicalThings = requires(T t)

{

{ t.Magical() } -> std::same_as;

};
```

Basically, within the concept block, we are defining what the concept's traits are. They could be methods, member properties, type constraints, and so on.

If you recall in our earlier example, we used -> showing we need a function to return and this example says that it is the same as void. This

looks a bit odd, but there was a clarification to the standard and decided to do things this way.

Previously it was possible to say -> int but there were cases where this was not desirable. You can still leave off the return type and just have Then, it does adhere to the function name, but it does not check the return type. So, it could be or and so on.

Recipe 1.12: Different types of expressions

Needing to be able to constrain a function's return type in different ways.

Concepts can constrain the return value, but it needs to be exact, or at least similar.

Speaking of signatures for requirements, let us consider the following snippet:

```
templateT>

concept HasDoubleIt = requires (T t) {

  // member function that returns the same type,

  // but still could overflow

  { t.DoubleIt() } -> std::convertible_to;

// similar to above, but is a static function

  { T::DoubleIt(t) } -> std::convertible_to;

  // just in case the large number is odd,
```

```
  // and we'd have a leftover fraction

  { t.HalfIt() } -> std::same_as;

  { t.SomeNoExceptFunction() } noexcept -> std::same_as;

};
```

Here are a few things to note:

The last SomeNoExceptFunction was added to note that in addition to the function name, function parameters, and return type - the compiler also checks on noexcept clauses. So, if you provided a type that had SomeNoceptFunction but did not have then it will not match.

For those who know the language, it does seem to be a bit Python-esque, where (in pure terms) it does not define an interface. However, if what you defined obeys the requirements of the interface definition, then it is considered that type.

Some of this relates to an odd acronym called Substitution Failure Is Not An Error The premise is that when the compiler is evaluating template type substitutions, in some cases, supplied arguments will not fit into the template definition. Instead of immediately failing with an error, the compiler will just go on and try another template. So, when trying a particular type to see if it adheres to the concept, failing the first time does

not mean that nothing will ever work. Rather, it just means that the template does not fit for that example.

To illustrate this, let us look at another snippet.

Recipe 1.13: Specializations

It would be good to have a mechanism that checks for the variables we need.

Concepts can define a constraint so that the proper variable exists.

Refer to the following code snippet:

```
#include

template T>

concept HasIngredientList = requires (T v) {

v.IngredientList;

};

template T>

concept CompleteRecipe = requires (T v) {

    v.IngredientList;
```

```
    v.Instructions;
};
void RecipeFunction(HasIngredientList auto x) {}
void RecipeFunction(CompleteRecipe auto x) {}
struct ThingWithIngredients {
    int IngredientList;
};
struct ThingWithSomethingElse {
    bool IsHealthy;
};
struct ThingWithIngredientsAndInstructions {
    int IngredientList;
    int Instructions;
};
```

```
struct AnotherThingWithIngredients {

    float IngredientList;

};

int main() {

    // compiles, since only one method is possible

RecipeFunction(ThingWithIngredients{});

    // compiler error, since the parameter doesn’t match any functions

    RecipeFunction(ThingWithSomethingElse{});

    // compiler error, since ambiguous

    RecipeFunction(ThingWithIngredientsAndInstructions{});

    // also compiles, even though member variable is a different type

    RecipeFunction(AnotherThingWithIngredients{});

}
```

So, here we have two concepts: one that has a member variable and another that has IngredientList as well as We have two overloaded functions named that take different arguments. We have then defined a few structures with various fields.

When you call a function with everything builds without issue. That is because ThingWithIngredients has and there was one function that accepted an IngredientList as an argument.

When we call with it cannot find any function that takes that as a parameter – so naturally it fails to compile.

The interesting scenario is when we call using Some might think it would be fine since that matches the requirements of the CompleteRecipe concept. The problem, however, is that it also still matches the function that accepts the concept and so, it is a compiler error saying it is ambiguous.

The other case, when we use an instance of also works fine. That is because we never specified the type for the member fields, but just the name.

To fix our ambiguous problem, we will have to make an adjustment to the concept requirements:

```
template T>

concept CompleteRecipe = HasIngredientList && requires (T v) {
```

```
    v.Instructions;

};
```

If we say that one concept is more specific than the other (but adding a requires clause), then the compiler can figure things out.

Note: Different to the preceding example where a requires expression might be embedded within a clause, you might see a requires requires clause. This would be used in something called an anonymous concept. Note that although it could be done, it is something that should not actually be done - it just seems safer to state them separately.

Some other things to keep in mind about concepts are as follows:

Concepts cannot be recursive.

They cannot be partially defined.

They can be redeclared. However, this is not normally desirable and thus it is something to be aware of.

Conclusion

This chapter showed you that concepts are an extremely important new feature of the C++ language. They have made using templates very accessible; it is a lot easier to understand when the compiler gives you an error, and a lot more flexible. We have shown there can be different parts of a concept, from the requires clause to all the different ways to state requires expressions.

In the next chapter, the book will show you that there are a lot of new concepts built into the language that you can use.

Points to remember

Concepts were a new C++20 feature.

Concepts will greatly enhance and make it easier for you to do templates.

There are different ways to define how you want to make a constraint.

References

CPPreference. (n.d.). SFINAE. Retrieved from https://en.cppreference.com/w/cpp/language/sfinae

Dargo, S. (n.d.). For different ways to specify a function Retrieved from https://www.sandordargo.com/blog/2021/02/17/cpp-concepts-4-ways-to-use-them

Grimm, R. (n.d.). For introduction to Concepts using Integral Retrieved from https://www.modernescpp.com/index.php/defining-concepts-with-requires-expressions

Toth, S. (n.d.). Was used as inspiration for the recipe with the Retrieved from https://itnext.io/c-20-concepts-complete-guide-42c9e009c6bf

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 2

Using the New Core Language Concepts

Introduction

Now that you have been introduced to the huge new feature of concepts, we will take a look at some pre-existing ones. Thankfully, you do not need to create all of your own from the very beginning – there are lots that have already been made for you to use, or as you just learned, you can reuse when creating your own.

This along with the next four chapters, will discuss pre-made concepts. This will let you know quite a few things. For instance, you will see that there are a lot of concepts that you can use (or reuse). The volume itself signifies just how big of a new feature concepts are. Moreover, since so many existing checks are being used in new concepts, you should understand what direction the C++ language is moving in.

Structure

In this chapter, we will cover the following recipes from C++20:

Recipe 2.1: same_as

Recipe 2.2: convertible_to

Recipe 2.3: derived_from

Recipe 2.4: integral, signed_integral, unsigned_integral

Recipe 2.5: floating_point

Recipe 2.6: assignable_from

Recipe 2.7: swappable mean

Recipe 2.8: destructible

Recipe 2.9: constructible_from

Recipe 2.10: default_initializable

Recipe 2.11: move_constructible, copy_constructible

Recipe 2.12: common_with and std::common_reference_with

There is good news relating to the compiler readiness for the features we will be talking about in this chapter. All seem to have been implemented by the three main compiler companies:

{

// works with when tested clang

// 20231029 yes yes yes

yes yes

y y y

// 20231029 yes

// 20231029 yes

// 20231029 yes

// 20231029 yes

// 20231029 yes yes yes

```
        // 20231029  yes

yes  yes

// 20231029  yes  yes  yes

// 20231029  yes  yes  yes

      // 20231029  yes

}
```

Objectives

Concepts are an extremely powerful new feature. We will be mentioning them in most chapters of this book. By the end of this chapter, the reader will start to learn some of the existing concepts they can use.

Recipe 2.1: same_as

Problem: We want an easy way of checking, to ensure that our object has the exact right type of member function we want to use.

Solution: C++ 20 has a new concept named

We have actually seen a couple of these new concepts already. We used them in the previous chapter to compare types.

The same_asT2> concept (released in C++ 20) is satisfied if T1 and T2 are the same type.

Let us make it easy and define all the headers we will use for this chapter:

```
#include

#include

#include

#include

#include

#include
```

#include

If we create some code that uses our classes, some work better than others:

template T>

= requires (T r) {

{ r.HasTheRightIngredient() } ->

};

{

{}

};

{

{}

};

{

};

{

{}

};

{

{}

};

: Recipe2A { };

: public Recipe2A { };

{

description, HasRecipeWithRightIngredient auto r) {

```
<< description << " has the right

}



{

<< << << __FUNCTION__ <<

// found the right ingredient

  Recipe2A{});

// COMPILER ERROR: has the wrong ingredient

//PrintRecipeWithRightIngredient(typeid(Recipe2B).name(), Recipe2B{});

// COMPILER ERROR: has a member variable of that name, but not a function

//PrintRecipeWithRightIngredient(typeid(Recipe2C).name(), Recipe2C{});

// COMPILER ERROR: has the correct member function name, but not correct type

//PrintRecipeWithRightIngredient(typeid(Recipe2D).name(), Recipe2D{});
```

```
// COMPILER ERROR: has the correct member function name, but type doesn't match

//PrintRecipeWithRightIngredient(typeid(Recipe2E).name(), Recipe2E{});

// COMPILER ERROR: is derived from class with that function, but not accessible

//PrintRecipeWithRightIngredient(typeid(Recipe2F).name(), Recipe2F{});

// is derived from class with that function, and is accessible


  Recipe2G{});

}

};
```

The code should compile without issue, and we get the following results:

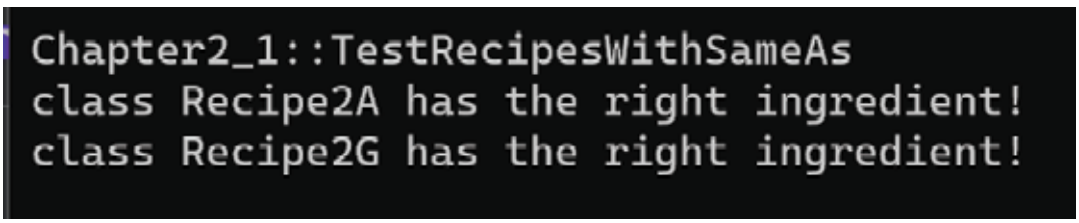

Figure 2.1: Output from TestRecipesWithSameAs

However, if we had kept those lines that are commented out, we get a lot of compiler errors, as shown in Figure

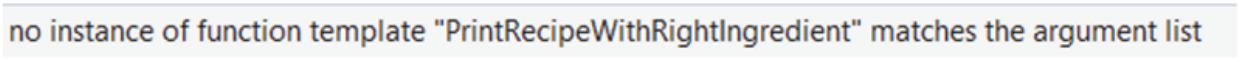

Figure 2.2: Compiler error

It shows that only the first and last were compiled properly (for Recipe2A and

We can understand it failing if the concept is looking for the method HasTheRight when we provide HasTheWrongIngredient Simultaneously, it can also be because we are looking for a method when the class has a member variable

Thus, we provide a class with the right function name, but the wrong type uses a bool).

Some might think that maybe it would work if we provided a function that can hold more than the int we are looking for has a but same_as fails if it does not match.

We are getting closer the but then, we have an accessibility problem – since we did not state to inheritance the access defaults to private.

You might have noticed something odd in the code – that it is passing the typeid's name into the function. In some cases, it is not easy to get the underlying type from a template, so to keep the code examples simple we just passed in the class name.

Lastly with we declared the inheritance correctly, and so it compiles.

Recipe 2.2: convertible_to

Problem: We want an easy way of checking, to ensure that our object has the method we can use. It does not have to be the same, but something close.

Solution: C++ 20 also has a new concept named

Looking at our previous example, there was at least one scenario where we would hope it could work, but same_as wants things to match completely. Another concept that was created was which says that if the expression can be implicitly and explicitly converted to the desired type (in the same manner), then it will satisfy the requirement. That caveat of in the same manner might be tricky in some edge cases, but for now, it is not something to focus on.

Now we will create a new concept that uses along with some new code. Then, we will get the following albeit different results:

```
template T>

= requires (T r) {

{ r.HasTheRightIngredient() } ->

};

{
```

```
description, HasRecipeWithAlmostRightIngredient auto r) {

<< description << " has the ingredient that is right, or close



}



{

<< << << __FUNCTION__ <<

// found the right ingredient

  Recipe2A{});

// COMPILER ERROR: has the wrong ingredient

//PrintRecipeWithAlmostRightIngredient(typeid(Recipe2B).name(),Recipe2B
{});

// COMPILER ERROR: has a member variable of that name, but not a
function

//PrintRecipeWithAlmostRightIngredient(typeid(Recipe2C).name(),Recipe2C
{});
```

```
// COMPILER ERROR: has the correct member function name, but not
correct type

  Recipe2D{});

// COMPILER ERROR: has the correct member function name, but type
doesn’t match

  Recipe2E{});

// COMPILER ERROR: is derived from class with that function, but not
accessible



//PrintRecipeWithAlmostRightIngredient(typeid(Recipe2F).name(),Recipe2F
{});

// is derived from class with that function, and is accessible

  Recipe2G{});

}

};
```

The methods that worked before and are still fine, since the exact same time still satisfies the requirement. The methods that we were pretty sure would not work: such as Recipe2B Recipe2C (because it is a variable versus a function), and (since the method is not accessible); still failed to compile.

For the case of which failed because the bool did not match the sought-after int - now compiles. That is because we can easily convert the bool to a long.

The curious case is of which now can convert a long to an Indeed, it can convert it, but keep in mind when using that it does not necessarily check for overflow.

Having at least some of the recipes work, we get the output:

```
Chapter2_2::TestRecipesWithConvertibleTo
class Recipe2A has the ingredient that is right, or close enough!
class Recipe2D has the ingredient that is right, or close enough!
class Recipe2E has the ingredient that is right, or close enough!
class Recipe2G has the ingredient that is right, or close enough!
```

Figure 2.3: Output from TestRecipesWithConvertibleTo

Recipe 2.3: derived_from

Problem: We want an easy way to check, to ensure our type is related to another type.

Solution: We can use the concept

We will look at the C++20 concept named derived_from to validate one type derives from another. To validate the converse constraint, we can use the C++11 type trait named Actually, the C++20 concept derived_from uses Moreover, there are differences that you should be aware of.

In this sample, we will start off defining our types:

```
: {};

: {};


: {};

{


type1Name, type2Name,
```

```
bool is1DerivedFrom2, bool is2DerivedFrom1, bool is1BasePf2, bool
is2BaseOf1) {

<< type1Name << (is1DerivedFrom2 ? " IS " : " is not

  << "derived from (or the same) as " << type2Name <<



<< type2Name << (is2DerivedFrom1 ? " IS " : " is not

  << "derived from (or the same) as " << type1Name <<

<< type1Name << (is1BasePf2 ? " IS " : " is not

  << "the base (or the same) as " << type2Name <<

<< type2Name << (is2BaseOf1 ? " IS " : " is not

  << "the base(or the same) as " << type1Name <<

<<

}



{

<< << << __FUNCTION__ <<
```

CreamySeafoodChowder>,

CreamySeafoodChowder>,

CreamySeafoodChowder>,

CreamySeafoodChowder>);

BaseSeafoodChowder>,

CreamySeafoodChowder>,

BaseSeafoodChowder>,

CreamySeafoodChowder>);

}

{

<< << << __FUNCTION__ <<

BaseSeafoodChowder>,

TomatoSeafoodChowder>,

BaseSeafoodChowder>,

TomatoSeafoodChowder>);

}

{

<< << << __FUNCTION__ <<

AnotherChowder>,

AnotherChowder>,

AnotherChowder>,

AnotherChowder>);

OtherChowderBase>,

```
AnotherChowder>,

OtherChowderBase>,

AnotherChowder>);

}

{

<< << << __FUNCTION__ <<









}

{

TastingCreamyChowder();

TastingTomatoChowder();
```

```
TastingAnotherChowder();

CheckingValueTypes();

}

};
```

And once we define all of our types, we can then start investigating them. We will use the new concept, along with the older type trait, to compare both ways. We will look at the portions of the output individually.

Unsurprisingly, when comparing the same types – it says that it is the same thing, as shown in Figure

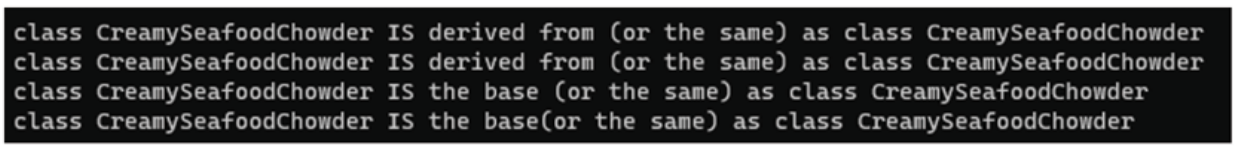
class CreamySeafoodChowder IS derived from (or the same) as class CreamySeafoodChowder
class CreamySeafoodChowder IS derived from (or the same) as class CreamySeafoodChowder
class CreamySeafoodChowder IS the base (or the same) as class CreamySeafoodChowder
class CreamySeafoodChowder IS the base(or the same) as class CreamySeafoodChowder

Figure 2.4: Output when comparing same types

When comparing a derived class to its base, it also acts as expected, as shown in Figure

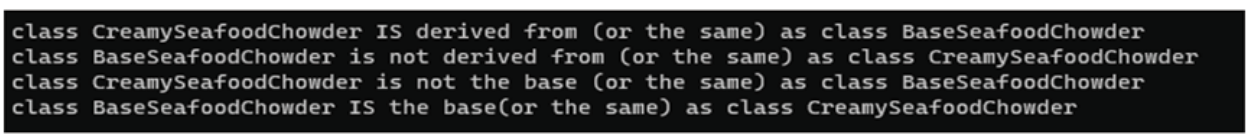
class CreamySeafoodChowder IS derived from (or the same) as class BaseSeafoodChowder
class BaseSeafoodChowder is not derived from (or the same) as class CreamySeafoodChowder
class CreamySeafoodChowder is not the base (or the same) as class BaseSeafoodChowder
class BaseSeafoodChowder IS the base(or the same) as class CreamySeafoodChowder

Figure 2.5: Output when comparing derived types

Next, we compare the TomatoSeafoodChowder and This gives the output shown in Figure

```
class TomatoSeafoodChowder is not derived from (or the same) as class BaseSeafoodChowder
class BaseSeafoodChowder is not derived from (or the same) as class TomatoSeafoodChowder
class TomatoSeafoodChowder is not the base (or the same) as class BaseSeafoodChowder
class BaseSeafoodChowder IS the base(or the same) as class TomatoSeafoodChowder
```

Figure 2.6: Output when comparing to private base type

This might not be what we expected since BaseSeafoodChowder is the base class. The problem however, comes back again to accessibility. It was not publicly derived, and so the concept check does not properly see the base type.

To move from classes to structs, we get similar and expected results, similar to if it was a class. Refer to the following Figure

```
struct AnotherChowder IS derived from (or the same) as struct AnotherChowder
struct AnotherChowder IS derived from (or the same) as struct AnotherChowder
struct AnotherChowder IS the base (or the same) as struct AnotherChowder
struct AnotherChowder IS the base(or the same) as struct AnotherChowder

struct AnotherChowder IS derived from (or the same) as struct OtherChowderBase
struct OtherChowderBase is not derived from (or the same) as struct AnotherChowder
struct AnotherChowder is not the base (or the same) as struct OtherChowderBase
struct OtherChowderBase IS the base(or the same) as struct AnotherChowder
```

Figure 2.7: Output when comparing structs

Lastly, we will try some fundamental/simple types. The following results shown in Figure 2.8 may not be as expected:

```
int is not derived from (or the same) as int
int is not derived from (or the same) as int
int is not the base (or the same) as int
int is not the base(or the same) as int

long is not derived from (or the same) as int
int is not derived from (or the same) as long
long is not the base (or the same) as int
int is not the base(or the same) as long
```

Figure 2.8: Output when comparing fundamental types

This shows us that we cannot use these two concepts or for simple types. Rather, we can use them only for structs and classes. It is worth keeping in mind that for simple types, you will need to still use same_as or convertible_to as discussed previously.

Recipe 2.4: integral, signed_integral, unsigned_integral

Problem: There should be an easy mechanism to constrain a value to an integer.

Solution: There were concepts (created in C++ 20) that allow you to constrain values.

If you recall in the previous chapter, we created our own concepts that constrained our templates to an integer. Thankfully, we do not need to start from nothing, and create that code since parts of that is also part of the standard. However, it is still important to understand, and a nice first step to take into the following three concepts:

An integer, which could be positive or negative.

An integer (normally marked with a u suffix) that cannot be negative. It has twice the range since a bit is not being used for the sign.

An integer that does not have a u suffix.

If we create a similar concept as before:

template T>

=

```
template T>

=

{



typeName, PostivePageCount auto pg) {

<< typeName << " is

}

{

<< << << __FUNCTION__ <<

// does not compile

//TryToPrintNumber("integer", 'A');

// does not compile
```

```
//TryToPrintNumber("negative integer", -1);

// does not compile

//TryToPrintNumber("number with decimal", 12.3);

}

};
```

We get the results:

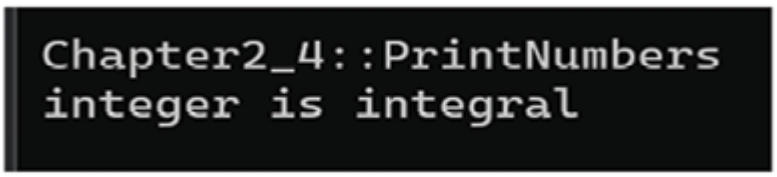

Figure 2.9: Results from PrintNumbers

Then similar to before, we could pass in a value that adhered to that constraint; such as 123u. However, if we pass in a value such as -123, 12.3, or 123 (that is not marked as unsigned), then we would get a compilation error.

Recipe 2.5: floating_point

Problem: We want an easy mechanism to constrain something to only allow a floating_point type.

Solution: Similar to the others mentioned, in C++ 20, there is a new floating_point concept.

Based on the last topic, you can guess that this concept is straight forward. It will constrain to floats and doubles (of various sizes), but not allow integrals. Refer to the following code sample:

```
template T>

=

{



typeName, LitresOfWaterRequired auto litres) {

<< typeName << " is floating

}
```

{

<< << << __FUNCTION__ <<

to decimal

// the ones below fail to compile

//TryToPrintFloatingNumber("negative integer", -1);

//TryToPrintFloatingNumber("char", 'a');

}

};

The results are:

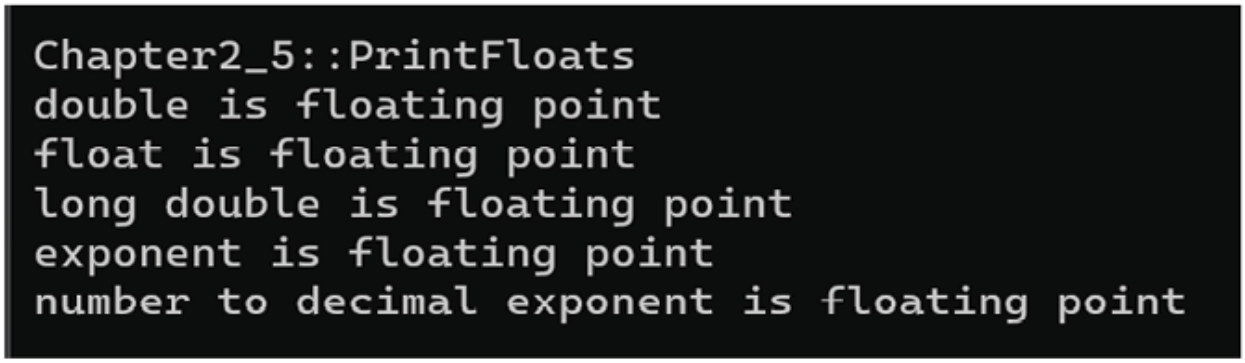
Chapter2_5::PrintFloats
double is floating point
float is floating point
long double is floating point
exponent is floating point
number to decimal exponent is floating point

Figure 2.10: Results from PrintFloats

As expected, the last couple of values (an integer and a char) fail the constraint for the function. Otherwise, it will allow floating point values of a variety of types.

You will probably know the usual suffixes of L and f (for long double and respectively), and e for exponents. In C++ 17, there was a new float-related literal p (or for integral power of 2. As 5e4 means 5 x we can also do 5p4, which means 5 x

Suggestion

This flexibility allows you to overload your functions, depending on the concept you specify as a parameter. Earlier, you needed a parameter that will hold the largest amount possible, and it is always doing a conversion (for example, if you pass a short int to a function that has a parameter of a double Using concepts allows you to keep its native value (as long as the function handled the various types), or you could have different functions (one for std::integral and another and have the logic act differently in each function. Using different functions allows you more control over the parameters and custom logic for each implementation, as shown:

```
typeName, auto number) {

// do some logic specific to integrals

<< typeName << " is an floating

}

typeName, auto number) {

// do some logic specific to floating point values

<< typeName << " is an
```

}

Recipe 2.6: assignable_from

Problem: We want to be able to check if we can assign a value of one type to another specified type

Solution: The concept assignable_from was created in C++ 20 to have a constraint, whether something is assignable.

The next set of recipes in this chapter does not relate to what the type is, but is more related to the capabilities of that type. Adding these constraints will make the code cleaner and safer so that you do not have to worry about those types of odd run time issues.

The concept assignable_from checks to make sure that the first template parameter is a proper left-hand-side recipient, and the second parameter is a proper right-hand-side value. In other words, we cannot try something equating to 123 = x; or 123 = 42;

Let us look at our example:

```
};

{



rs) { rightValue = rs; }
```

```
};

};

template T>

concept AcceptableForLeftSideRightSide = T>;

{


ls, AcceptableForLeftSideRightSide auto rs) {


<< << " can be given a " << <<

}


{

<< << << __FUNCTION__ <<


LeftSide ls { rs };

TryToAssign(ls, rs);
```

```
// Compiler fails due to not assignable

// ls, os);

}

};
```

And we have the output:

```
Chapter2_6::AssignSomeSides
struct LeftSide can be given a struct RightSide
```

Figure 2.11: Results from AssignSomeSides

We will see that we have defined a concept for checking the assignment, and it works when we try to assign the RightSide to the but there is a compiler error if we try to assign the since that has not been defined as acceptable.

Suggestion

At this point, you might be thinking you can get rid of a lot of unit-test code since the concepts will cover so much that there is not much point in testing anything. Although concepts are very powerful and might reduce parts of your code-base, hoping to have very little test code would be far from realistic. There are still many cases where you would need to have a unit-test code to check something at runtime. Moreover, an important aspect to note is that having your classes/types compiled properly to the concepts' constraints does not mean that everything will work fine. The compiler will accept if you try std::assignable_frombut just because you can assign a double value to an integer variable – does not mean you wanted to do that. There is not much boundary/overrun protection on these concepts. We would even have to look up the rules to say whether we give consistent rules for these situations, and might need to ask questions such as, does it throw an overrun exception? Does the target variable wrap Does it stay at the target's maximum value? Perhaps in a future C++ release, there will be new concepts that do this checking, but for now, the safest thing is to build off of these handy starter concepts and suit them to your logic, based on what the functionality is. Then you can write your unit-tests to test that functionality you worked towards, to ensure it does what you intended.

Recipe 2.7: swappable

Problem: It would be good to have an easy way to check if you can swap two values.

Solution: The C++ 20 library has the concept

We just looked at the concept assignable_from that allowed you to set the left-hand side from the right-hand side. Now, we will look at which in a way is a double It might have been conceived from the C++17 template, called and it is implemented on Ranges. However, we will focus on them in another chapter. The greatest benefit of this new concept is that it ensures equality preservation (assigning a value does not guarantee preserving the value). Since you are trying to swap two values of the same type – in most cases, you code should handle this, but it is good to have this extra bit of assurance.

The concept std::swappable has one templated type and checks whether two variables of the same type can swap values. We also have which can check if two variables of different types can swap values.

Let us look at an example of both and

```
// Reusing From PreviousSection

};
```

```
{


rs) { rightValue = rs; }

};

};

template T>


concept AcceptableToSwapWithLeftSide = T&>;

{


ls, AcceptableToSwapWithLeftSide s) {

<< << " can be swapped with a " << <<

SwapSides(ls, s);

}
```

```
{


rs };









// does not compile, because we cannot swap

// PerformSwap(ls, rs);

// does not compile, because we cannot swap, even though both empty

// SwapSides(rs, os);

}

template T2>

t1, T2&& t2) {

auto holder =
```

t1 =

t2 =

}

};

Recipe 2.8: destructible

Problem: We want a mechanism to tell us if instances of our type can be destroyed when finished with.

Solution: There is a concept named destructible (released with C++ 20).

In the C++ 11 release, they defined a few checks:

is_destructible

is_trivially_destructible

is_nothrow_destructible

Although they have been around for a while, we will just touch upon them, so that you understand what the C++ 20 concept requires.

The check/test is_destructible does more, but the short answer is that it checks if something can be destroyed. It would fail for conditions such as:

If you explicitly say the destructor is not defined, ~MyType() = delete;

If you have a static type, it does not get destroyed until the program terminates.

Next would be is_trivally_destructible, meaning that it has a default destructor. This default destructor could be undefined, or simply ~MyType() = If your destructor is marked with then trivial would be false.

Lastly is is_ which would be true in most cases, although false if you set your destructor to That sounds like an odd thing to do, but the easiest scenario would be a singleton – so that it cannot accidentally be deleted.

This can be summed up in the following Table

Table 2.1: Comparison of destructible checks

This review is a long lead-in to simply say that the std::destructible concept uses the std::is_nothrow_destructible check (or technically, it uses

Thus, following these rules from the preceding table, if we define some structs, and add some code to verify what we have been talking about, we get the following code snippet:

```
{

};

{

~SimpleRecipe_default() =

};

{


~SimpleRecipe_simple() {}

};

{

~SimpleRecipe_noexcept() noexcept {}

};
```

```
{

~SimpleRecipe_deleted() =

};

template T>

=

typeName, CannotFailRecipe auto recipe) {

<< "Recipe " << typeName << " cannot be messed

}

{




{

<< << << __FUNCTION__ <<

SimpleRecipe_undefined{});

SimpleRecipe_default{});
```

```
SimpleRecipe_simple{});

SimpleRecipe_noexcept{});


// will not compile since it does not pass the concept's constraint

// CheckIfRecipeCannotFail("deleted", SimpleRecipe_deleted{});

}

};
```

Note that our last method fails to compile because the type is defined with a destructor = deleted;

Aside from that issue, we have the output:

```
Chapter2_8::PrintCannotFailRecipes
Recipe undefined cannot be messed up
Recipe default cannot be messed up
Recipe simple cannot be messed up
Recipe noexcept cannot be messed up
```

Figure 2.12: Results from PrintCannotFailRecipes

Recipe 2.9: constructible_from

Problem: Needing to know that a type can be initialized, given a particular set of arguments.

Solution: C++ 20 created a concept named

Based on the C++11 is_constructible template, the new concept enables you to have a constraint, based on the argument types you plan to pass into the constructor.

The reason why we discussed destructible before constructible_from is that constructible_from requires This logically makes some sense, because to be constructible, you also need to be destructible.

As before, we start off with some struct definitions using the following code, and then the implementation will check those structs.

```
{

};

{

SimpleRecipe_no_args_default() =
```

```
};

{

SimpleRecipe_no_args_simple() {}

};

{

SimpleRecipe_no_args_noexcept() noexcept {}

};

{

SimpleRecipe_static() =

SimpleRecipe_static(SimpleRecipe_static&& other) =


arg1) {


}
```

```
{}

};

{

arg1) {}

other) =

SimpleRecipe_one_arg_simple() =

};

{

arg1, float arg2) {}

other) =

SimpleRecipe_two_args_simple() =

};

typename T0>

auto recipe) { }
```

```
{




{

<< << << __FUNCTION__ <<









// ones below fail to compile

//CheckRecipeWithNoArgs(SimpleRecipe_one_arg_simple(1) );

//CheckRecipeWithNoArgs(SimpleRecipe_static::CreateRecipe("static"));

//CheckRecipeWithNoArgs(SimpleRecipe_two_args_simple(1, 2.0f));

}
```

```
};
```

In this code sample, most of the simple structs we defined met the requirement that they can be created without any arguments. The two cases where it does not compile are:

The type that can only be statically created.

The case where the constructor has arguments.

In some cases, you might still use (non-concept) checks for how the type is being created:

std::is_trivially_constructible_vT1, …>

std:: is_nothrow_constructible_vT1, …>

std:: is_constructible_vT1, …>

However, it will be handy in some cases where you want to constrain your types.

Recipe 2.10: default_initializable

Problem: We want a way to check if a type has default initialization.

Solution: The concept default_initializable (in C++ 20) meets this need.

In C++, default initialization occurs when there is no initializer. Sometimes the variable's value may not be determinable – so you cannot (or rather, should not) use it until it is properly set. Some other languages have clearer rules about initializing unspecified variables to NULL or 0, but C++ does not have that defined.

Various types can be initialized differently;

value-initialized; MyType()

default-initialized: MyType t;

direct-list-initialized: MyType { }

This concept requires std::constructible_from> as well as the default initialization, such as MyType { }

If we look at the code example:

```
{
```

```
};

{

TypeWith_DefaultConstructor() =

};

{

values = { "great" };


args) { values = args; }

};

{

TypeWith_DefaultInitialization() {}


};


{

  value = valueGiven;
```

```
}

};

{

v) {

<< "Type " << << " is default initializable" <<

}

v) {

<< "Type " << << " is NOT default initializable" <<

}

{

<< << << __FUNCTION__ <<
```

```
TypeWith_ListInitializing


TypeWith_Param 5 };

PrintType(t1);

PrintType(t2);

PrintType(t3);

PrintType(t4);

PrintType(t5);

}

};
```

We have created a handle of classes, all of which are initialized differently. When we see the output:

```
Chapter2_10::LookAtDefaultInitializables
Type struct TypeWith_ImplicitDefaultConstructor is default initializable
Type struct TypeWith_DefaultConstructor is default initializable
Type struct TypeWith_ListInitializing is NOT default initializable
Type struct TypeWith_DefaultInitialization is default initializable
Type struct TypeWith_Param is NOT default initializable
```

Figure 2.13: Output from LookAtDefaultInitializables

We can see the types that need explicit parameters in the constructor fail this concept.

Recipe 2.11: move_constructible, copy_constructible

Problem: There are times when you need to know if a type has an overload that is a copy or move constructor.

Solution: In C++ 20, the move_constructible and copy_constructible concepts were added.

You can see a pattern forming here. The move_constructible that was added in C++20 was based on the C++ 11 check is_move_constructible (and its variants). Similarly, the concept copy_constructible was based on the check Both are very important but for different reasons.

You need to know about the copy constructor because it is used more often than you might think about in the following situations:

When you pass an existing instance to a constructor of the same type.

When you call a function and do not pass your type's object by reference – then, it will call the copy constructor to use the local copy.

When you return an object from a function, it will also create a copy.

Copy constructors are very important. In their absence, you would be doing a shallow copy – which can lead to unintended outcomes of code

updating two The other bad outcome would be, if in case you delete the shallow copy, and it is not done properly, you are freeing the memory and leaving a dangling pointer that the other copy still thinks is valid. Before we talk further about concepts, let us first review with an example showing some dangers of copy constructors and not using arguments correctly.

In this example, we will define some new types, and then look at the steps for a recipe:

```
{



_steps;



// primary constructor

steps) {

<< "in constructor with list of

  _steps =

}

// copy constructor
```

```
obj) {

<< "in constructor with

  _steps =

  obj._steps->begin(), obj._steps->end());

}

~RecipeSteps() {

<< "in

delete

}

recipe) {

<< "Steps for

for step = recipe._steps->begin(); step != recipe._steps->end(); ++step) {



<< "Step: " << *step <<

  }
```

```
}

title, RecipeSteps recipe) {

<< "Name of recipe:  " << title <<

PrintSteps(recipe);

}

step) { }

step) { _steps->insert(_steps->begin(), step); }

{ return _steps->size(); }

};

{



{

<< "simple recipe

RecipeSteps simple = RecipeSteps{ "eat" };
```

```
RecipeSteps::PrintSteps(simple);

<< "done simple

}

{

<< "\n\nmore difficult recipe

RecipeSteps simple = RecipeSteps{ "eat" };

RecipeSteps tougher = RecipeSteps(simple);



difficult tougher);

<< "done more difficult

}

{

<< << << __FUNCTION__ <<
```

```
DoSimple();

DoMoreComplicated();

};

};
```

The output as seen in the following Figure will show that we did something we probably did not want to do:

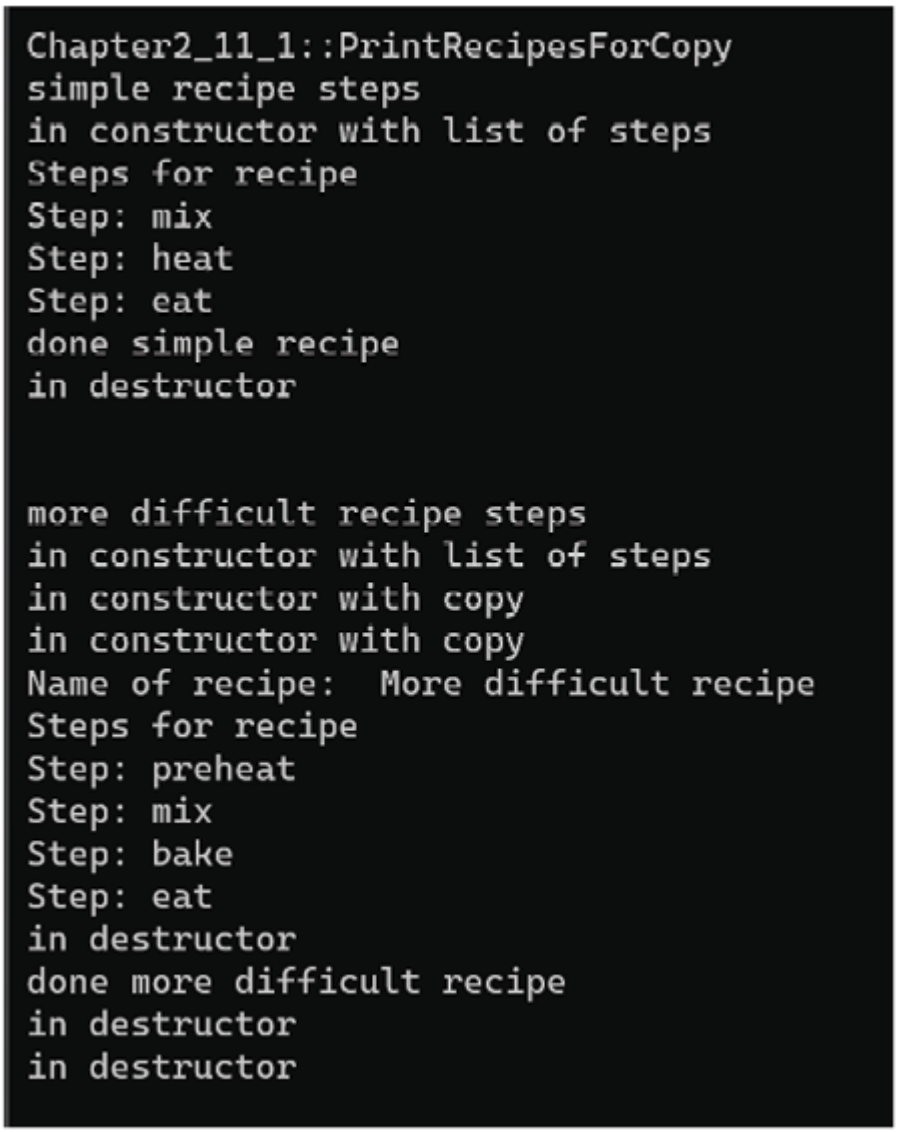

Figure 2.14: Output showing a problem in PrintRecipesForCopy

We realize that our PrintStepsWithTitle method does not use a reference or a pointer, which means it will create another copy that is passed into the function.

At least in this scenario, it would have been fairly painless: we are not modifying the object when printing (which implies that we should have added the const qualifier on the argument or the method), and at least we

created a copy constructor that did a deep copy. If we did not create that type of constructor, and we were modifying the object – then we would have lots of runtime issues.

There are certainly times when you might want to do a shallow copy, and also sometimes when you want to create a copy to pass into a function. You just need to be aware you are doing it, and be aware of the repercussions.

Conversely, there are times, when we want to ensure we only work on a copy of whatever was passed in. This would be a good reason to ensure we are using something that is In the following code example, we create a simplistic concept named HasListOfStepsTaken that has the function. Then we can use that in a concept we call NeedToMakeCopy that ensures it is Then we can have a function that has a parameter of and returns what we worked on.

Our revised code looks like:

```
template T>

= requires (T t, s) {

t.AddStep(s);

};

template T>
```

```
concept NeedToMakeCopy = HasListOfStepsTaken && requires (T v) {



};

{




{

<< "simple recipe

RecipeSteps simple = RecipeSteps{ "eat" };

RecipeSteps::PrintSteps(simple);

<< "done simple

}

{

<< "\n\nmore difficult recipe

RecipeSteps simple = RecipeSteps{ "eat" };
```

```
RecipeSteps tougher = RecipeSteps(simple);



difficult tougher);

<< "done more difficult

}

chapter2) {




}


{

<< "\n\nEnsuring recipe is

"eat" };

= DoSomethingWithCopy(existingRecipe);
```

```
<< "# Steps taken for recipe given = " <<
existingRecipe.GetNumberOfSteps() <<

<< "# Steps taken for recipe received = " <<
trackingRecipe.GetNumberOfSteps() <<

}

};
```

Then we can see in our output below, that the returning value was modified, but the value we passed in was not altered.

```
Ensuring recipe is copied
in constructor with list of steps
in constructor with copy
in constructor with copy
in destructor
# Steps taken for recipe given= 3
# Steps taken for recipe received = 4
in destructor
in destructor
```

Figure 2.15: Expected output from EnsureWorkingOnCopy

Recipe 2.12: common_with and common_reference_with

Needing a type that is common among other types.

Solution: There are new (C++ 20) concepts that you can use named common_with and

You may not use these too often. However, it does allow you to ensure that there is a type common to both classes you provide for the template.

common_with is a concept that says, given two classes and there will be a common type C that both can be converted to.

If we look at our small code sample, we are checking with various types if there have something in common.

```
T1, typename T2>

{

= T2>;

<< "Type: " << << (inCommon ? " does" : " does << " share a common type with " << <<

}
```

{

<< << << __FUNCTION__ <<

}

T1, typename T2>

{

= T2>;

<< "Type: " << << (inCommon ? " does" : " does << " share a common reference type with " << <<

}

{

<< << << __FUNCTION__ <<

```
}

{




{

LookAtCommonWith();

LookAtCommonReferenceWith();

}

};
```

And the results of which is:

```
LookAtCommonWith
Type: int does share a common type with char
Type: bool does share a common type with long
Type: bool does not share a common type with class std::bad_exception
Type: class std::tuple<int,long> does share a common type with struct std::pair<int,long>
```

Figure 2.16: Results from LookAtCommonWIth

It is interesting, and good to note, that a tuple and a pair have something in common. This was part of a C++23 enhancement.

Before we talk about the next concept, we will touch upon another C++20 feature named The first use that comes to mind relates to iterators – which we will talk a lot about in a future chapter, but it is the easiest way to describe the function and the concept. There is now a defined relationship between the reference part of an iterator (when you use the iterator's and the value type that is being held in the container, of which we are iterating. This will most likely be used only in special scenarios when dealing with iterators. The concept common_reference_with checks whether the reference and value-type relationship is, or can be, valid.

common_reference_with is similar to with the added constraint that the conversion to the common type must preserve equality. This means the conversion of equal inputs should result in equal outputs.

The author of the concept suggested that it might be used in a pair scenario, or some mapping where there is a relationship between two types. However, before these concepts, it was not easy to define the way they were related. Plus, these concepts simply tell you whether or not two types do have a compatible common type. It does not though, tell you what that type is. The C++11 std::common_type may be a more useful tool at times, since this will actually tell you the resulting type, if possible, for the two or more types provided.

We can see the second part of our output looks like:

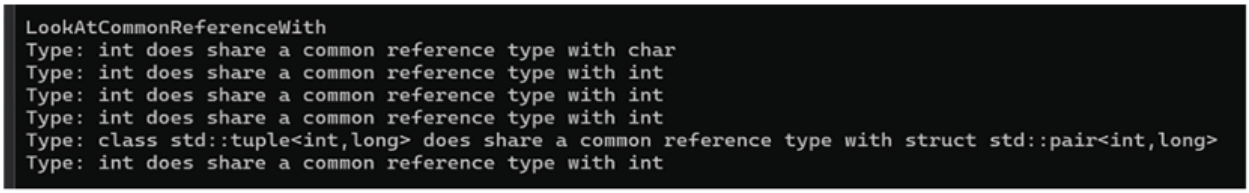
LookAtCommonReferenceWith
Type: int does share a common reference type with char
Type: int does share a common reference type with int
Type: int does share a common reference type with int
Type: int does share a common reference type with int
Type: class std::tuple<int,long> does share a common reference type with struct std::pair<int,long>
Type: int does share a common reference type with int

Figure 2.17: Results from LookAtCommonReferenceWIth

Conclusion

At the beginning of the chapter, we mentioned easier, cleaner, and Hopefully, you see the benefits of using some of the concepts we have mentioned so far. As with good class design, defining concepts is worth the effort. At the very least, you are self-documenting what the intent is. The intent shows how it was designed to be used, how it is currently being used, how the author wanted it to be used, and how it was tested to be used. As a code author, you do not want to bear the burden of the issues when a user goes outside the designed usage – so the best thing you can do is be as explicit as you can about how you intended that code to be used. Concepts will allow you to have compile time checks to ensure that the types you want to use will work with the code that uses them. After reading this chapter, the reader should understand that there are a lot of existing template concepts that can be used to validate the type of argument or its capabilities.

Points to remember

A lot of predefined concepts were created in C++20, based on checks that were introduced in C++ 11.

The concepts discussed in this chapter help you know how a type can be constructed, or how it was constructed, and some of its other capabilities.

Using these concepts will improve the quality of your code-base, and at the very least, the constraints show the intent on how the code is to be used.

References

Default_initializable (n.d.). Retrieved from https://en.cppreference.com/w/cpp/concepts/default_initializable

Common_reference (n.d.). Retrieved from https://stackoverflow.com/questions/59011331/what-is-the-purpose-of-c20-stdcommon-reference

CHAPTER 3

Using the New Comparison Concepts

Introduction

So far in this book, we have looked into what is a concept, how one creates a concept, and also started to look at some of the concepts that have been created in the standard library. The concepts we have looked at so far, help tell you what a type is, and how it is created. We will continue our journey with concepts, and this chapter is about comparison concepts. These concepts will help us understand the equality of two objects, and perhaps you can foresee that these concepts are used in several other concepts in future chapters.

Structure

In this chapter, we will cover the following recipes:

Recipe 3.1: equality_comparable

Recipe 3.2: equality_comparable_with

Recipe 3.3: totally_ordered

Recipe 3.4: totally_ordered_with

Recipe 3.5: three_way_comparable

Recipe 3.6: three_way_comparable_with custom types

Recipe 3.7: three_way_comparable_with

The good part of all of the features in this chapter, is that they all seem to be implemented by the three main compilers:

```
Chapter3_5_2::CompareTemperatures();

};
```

Objectives

By the end of this chapter, the reader will understand more about the comparison concepts that you can use and reuse, in the concepts made by you. These comparison concepts can be used by themselves, or used in conjunction with other concepts to define something more elaborate. We will see that they help to clarify a constraint, and in some cases, they help cut down a lot of code that you would normally have to write.

Recipe 3.1: equality_comparable

Problem: We want an easy way of checking if a type has a way of comparing if two similar objects are equal.

Solution: C++ 20 has a new concept named

Looking at some of the concept constraints, you can understand that equality is a lot trickier than you would first think. equality_comparable only used one type, but it does not dictate they have to be the same type – in other words, it could be a derived type as well. This is possible as long as all 4 equality checks between T and T b are met, meaning that the result of a == a != b == b != have to give a bool result. It would seem rare that somebody would design it to do something otherwise, but that is why we have these checks – just to be sure.

First, we will state what headers we will use for this chapter:

```
#include

#include

#include

#include
```

```
#include

#include

#include
```

Now, we will show the implementation. Naturally, we would normally put a lot of this code into header files, but for ease of reading it is all in the implementation. Our classes denote various types of rice, or some other small edibles. Some of them have an equality operator. When we look at our implementation, we will see some familiar issues:

```
{



Rice& other) const { return PrettySmall == other.PrettySmall; }

};

: Rice

{
```

```
BasmatiRice() { PrettySmall = }

};

: public Rice

{


LongGrainRice() { PrettySmall = }

};


{



OrzoPasta() { PrettySmall = }

Rice& other) const { }

};


{
```

```
AnotherTypeOfRice() { PrettySmall = }

AnotherTypeOfRice& other) const { }

};

{


food)

{

// ...

}



{

<< << << __FUNCTION__ <<
```

```
  Rice r;




CookTheRice(r);

// does not compile, since it does not meet constraint

// does not publicly derive from rice, so equality operator not visible

// CookTheRice(br);

CookTheRice(lgr);

// does not compile, since it does not meet constraint



// does have a public comparer function, but is comparing itself to Rice, not itself.

// CookTheRice(op);

CookTheRice(ator);

//static_assert(std::equality_comparable_with< BoxOfRice, BagOfRice>);
```

```
}

};
```

To cut down on trying to validate a lot of details, our CookTheRice function accepts something that has an

In the case where we simply pass in the Rice object, everything is fine. You will note in our header, for BasmatiiRice it is derived from but we forgot to say it is publicly accessible, and although it is derived from since the compiler cannot see the equality operator – it fails to compile. Our LongGrainRice object compiles fine, and you will notice in the header the equality operator is comparing itself to a Rice object. Since LongGrainRice derives from and has a PrettySmall variable, although LongGrainRice and normal Rice will never be the exact same thing, they meet the requirement that we can compare them. The next type we will try is and although it looks (a bit) like rice, and it has the equality operator – it fails to compile. It too, is comparing itself to but since it is not derived from this time we cannot say it meets the compiler requirements. The compiler knows they are not similar enough, so that constraint will fail. Lastly, we have which does not derive from but it has an equality operator, which compares to itself. These meet the requirements, so it compiles fine.

While we have other concepts that just check the type, or just check for the existence of a function or variable, this shows that we can also check for the existence of an operator. The compiler does not have concepts for

all the various operators, but it easy to say that the equality operator is the most important.

Recipe 3.2: equality_comparable_with

Problem: We want an easy way of checking if a type has a way of comparing if two different objects are equal.

Solution: equality_comparable_with was included in C++ 20.

Without going into the full concept definition, equality_comparable_withT2> uses and therefore by extension, equality_comparable and among some other checks. Basically, it looks for a common type that both T1 and T2 can be converted to, using that we discussed in an earlier chapter. You can see how various concepts are used together to make something even more powerful.

Thus, if your types T1 and T2 can be converted to a TC common reference, then you will be able to use the concept

Recipe 3.3: totally_ordered

Problem: We need a way of comparing a type, not just for equality, but for relative equality as well.

Solution: totally_ordered (in C++ 20) provides the comparison ordering needed for sorting.

We have just seen with equality_comparable that it is handy to have to know that you can compare two similar objects. The challenge though, is that you can compare them – but not sort them. Hence, the need for that in addition to the operator it also ensures there is the comparison operators for the following:

<

>

<=

>=

This means that you can get the relative ordering in a collection of objects.

Now we will switch dishes. In this preceding code snippet, we have defined a base that has a name and a width measurement. We have also

defined a number of comparison operators. Our implementation code defines the operators:

```
{



{




name;



nameGiven, int widthInMillimetres) : name(nameGiven),
width(widthInMillimetres)

  {

  }

const { return width; }

const { return name; }

const {

<< << " is " << << "mm
```

```
  }

};

: public Chapter3_3Food::Pasta {


Lasagna() : {}

};

: public Chapter3_3Food::Pasta {


Linguine() : {}

};

: public Chapter3_3Food::Pasta {


Spaghetti() : { }

};

};
```

```
== lhs, const Chapter3_3Food::Pasta& rhs)

{

== rhs.GetWidth();

}

Chapter3_3Food::Pasta& lhs, const Chapter3_3Food::Pasta& rhs)

{

< rhs.GetWidth();

}

<= lhs, const Chapter3_3Food::Pasta& rhs)

{

<= rhs.GetWidth();

}

Chapter3_3Food::Pasta& lhs, const Chapter3_3Food::Pasta& rhs)
```

{

> rhs.GetWidth();

}

>= lhs, const Chapter3_3Food::Pasta& rhs)

{

>= rhs.GetWidth();

}

{

{

<< << << __FUNCTION__ <<

=

```
pastaCount> pastas

{ Chapter3_3Food::Linguine{}, Chapter3_3Food::Lasagna{},
Chapter3_3Food::Spaghetti{} };

pastas.end());

for p : pastas) {

p.ShowInfo();

  }

<<

}

};
```

We also have a testing method that will create an array (of the base type) and add the different varieties. To be safe, we also do little checks to ensure that the type is and (our current focus) We then sort the array, and iterate through our array to print out our pastas.

We see in our output shown in the following Figure that things were sorted nicely.

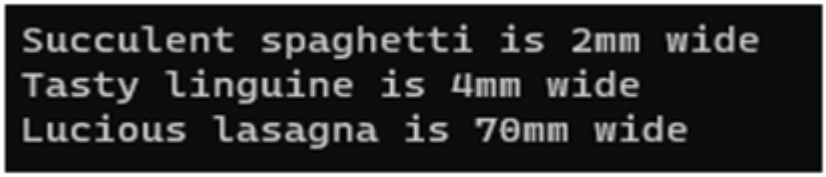

Figure 3.1: Output from TasteThePasta

Note that our code does not tell the array how to sort; the array will sort based on the comparison operators that are provided.

Suggestion

As an aside, you might have noticed that we did something different in our pasta code. In the Rice class, the == operator was right in the code (called the member-approach), while the Pasta == operator was outside (called the non-member approach). Perhaps there is a difference in comparing, to see if two objects are identical, versus comparing two objects for sorting purposes. However, in our case, all of the comparison operators were switched to the non-member approach. That was done to comply with the C++ Core Guidelines which says that for symmetric operations, use non-member functions (rule C.161).

We cannot say if that will ever turn into a compiler error that is enabled by default, and we will not attempt to go into any detail about why – just a note about these guidelines and why the new code was written differently. That rule also applies to our typical operators, if we use the three-way comparison operator, that we will learn about later, then we think it is acceptable to have as a member function.

If you think about it, there could be multiple ways to sort an object; whether it was by pasta width, or just alphabetically. Having the comparison operators within the class focusing on one item would seem to limit its abilities. One idea mentions to define an entity that contains multiple ways of comparing/sorting. That way your operators are still close together but you would then have the option to sort on different properties of your object.

Recipe 3.4: totally_ordered_with

Problem: We need a way of comparing multiple types, not just for equality, but for relative equality as well.

Solution: C++ 20 also contains

Just as equality_comparable_with uses equality_comparable for comparing equality on two different types, totally_ordered_with also does the various comparison methods. However, unlike it works on two different types. It will compare the types T1 versus T2, knowing there is a common TC type they both can use.

Recipe 3.5: three_way_comparable

Problem: We want to be able to do the comparison operators with less code.

Solution: C++ 20 created a concept named three_way_comparable that checks for the three-way comparable operator (but many will refer to it as the spaceship

Now that we have discussed equality_comparable (that is related to equality between two objects of the same type) and totally_ordered (that is related to comparison operators between two objects of the same type), we will talk about a concept related to a new operator – which performs a similar task.

In C++ 20, they created a new operator which by the looks of it, you can see why it is nicknamed the spaceship operator. This performs the normal comparison operations, but it does them altogether. It is a bit like a simple strcmp(s1, s2) where the result would be < or > Since it could be doing multiple comparison operations at the same time – this should speed up sort times, but mostly reduce the code, otherwise, you would need to create multiple comparisons. It is included in a new header which defines the operator, the concept, and a few other comparison classes that we will discuss later.

If we look at a quick example:

```
{



{

<< << << __FUNCTION__ <<

=

=



<=> hotOven) <

}

};
```

This shows the result of the three-way comparison is less than 0.

Another thing that is present in the header, are some utility functions:

std::is_eq : meaning equal, == 0

std::is_neq : meaning not equal, != 0

std::is_lt : meaning less than, < 0

std::is_lteq : meaning less than or equal, <= 0

std::is_gt : meaning greater than, > 0

std::is_gteq : meaning greater than or equal to, >= 0

We can redo our little bit of code using these utility functions:

```
{



{

<< << << __FUNCTION__ <<

=

=

<=> hotOven) <

<=> hotOven));

= <=> hotOven);

if {
```

```
<< "values are

  }

//

if {

<< "values are not

  }

//

if {

<< "first value is less than the

  }

//

if {

<< "first value is less than or equal to the
```

```
  }

//

if {

<< "first value is greater than the

  }

//

if {

<< "first value is greater  than or equal to the

  }

}

};
```

Running it, (aside from the asserts still working), we will get the output shown in Figure

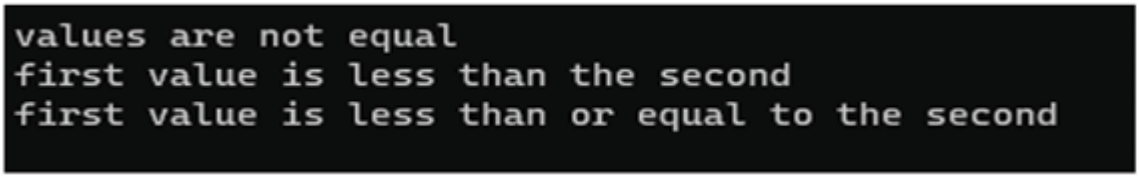

Figure 3.2: Output from CompareTemperatures

This makes sense when comparing the values 200 and 450.

Suggestion

These functions are not required, but might be helpful in some situations. If you do not use the helper functions, then it is advisable to use the comparisons mentioned earlier: == != > >= < What you do not want to do is to compare to == -1 or == which cannot be guaranteed to be the standard for all the code out there. We can say it is not standard, because the three-way comparison operator can be used on anything, and so it is not just a strcmp result but any third-party types that have been written. Some might be some might return a large negative number. You just have to be careful regarding what you are dealing with, and so it is safer to deal with values relative to 0, instead of hardcoded values. Hopefully, these utility functions can help you.

The next facet of this topic relates to the result type of a three-way comparison. In our example, we hid the type by declaring it There are several new types declared in the header, and in our sample code, the actual type is The list of possible types are as follows:

This is the result type of a three-way comparison that met the following:

Has all 6 relationship !=, <, <=, >, >=

Implies substitutability, describes as if a == == so equivalent values must be

Does not allow incomparable values; only one of: or > can be true.

Contains constants that show the result of a comparison:

less

equivalent (or equal)

greater

This is a lower ordering, which:

Has all 6 relationship !=, <, <=, >, >=

May not meet substitutability.

May admit incomparable values, and so could be none ==, or That would be the case if one of them a Number

Contains constants that show the result of a

less

equivalent

greater

unordered

This is the lowest ordering (aside from void meaning not possible) and it has the following features:

Has all 6 relationship !=, <, <=, >, >=

Does not meet substitutability.

Does not allow incomparable values.

Contains constants that show the result of a comparison:

less

equivalent

greater

Actually, you can see when comparing 24 <=> the result is and so, you get the type of result and the relative comparison result.

Moreover, in the header, we have the class which would indicate the strongest ordering result for a given type. If you are doing a three-way comparison with then you would get If comparing then you would get That is because doubles may not keep the exact same decimal value when acted upon (in other words, someDouble / 2.0 * 2.0 may not exactly equal

Thus, if you are creating library code that can compare different types, or types that are unknown at the time, then you might want to use this. If you are comparing multiple types, that would result in different orderings, then you might use common_comparison_category – which is the sort of the least common denominator for the different orderings. For all of your types, if all of the orderings were then the common_comparison_category would be If at least one of your orderings was not valid, then the common result would be Otherwise, it could be partial_ordering or

We have talked a lot about an advanced feature, but we have not talked about the concept yet. The short answer is the concept three_way_comparable checks if your type can perform a three-way comparison. Similar to how equality_comparable checks if the type has the operator ==, this concept will check if the type has the operator <=>. We will look into that next.

Recipe 3.6: three_way_comparable with custom types

Problem: We want be able to do comparisons with custom types.

Solution: Just like the <=> operator can work on simple types, you can also have it on more complex types, and sometimes very easily too.

Let us go back and modify some of our Pasta code. We will replace most of the comparison operators, with a single three-way operator. There are quite a few intricacies with this three-way operator, which we will talk about later, but for now we will say it is best to also still have an equality operator as well.

In our function definitions, we have not changed anything, except adding the three-way comparison function. This can be seen as follows:

{

{

name;

```
nameGiven, int widthInMillimetres) : name(nameGiven),
width(widthInMillimetres)

  {

  }

const { return width; }

const { return name; }

const {



<< << " is " << << "mm

  }

Pasta& rhs) const {

<=> rhs.GetWidth();

  }

== Pasta& rhs) const {

== rhs.GetWidth();
```

```
  }

};

: public Chapter3_6_1Food::Pasta {


Lasagna() : {}

};

: public Chapter3_6_1Food::Pasta {


Linguine() : {}

};

: public Chapter3_6_1Food::Pasta {


Spaghetti() : { }

};

};
```

```
{



{



<< << << __FUNCTION__ <<




=

pastaCount> pastas

{ Chapter3_6_1Food::Linguine{}, Chapter3_6_1Food::Lasagna{},
Chapter3_6_1Food::Spaghetti{} };

pastas.end());

for p : pastas) {

p.ShowInfo();

  }

<<
```

```
}

};
```

The result is similar to before, as can be seen in the following Figure even though we removed a lot of code for the other comparison functions:

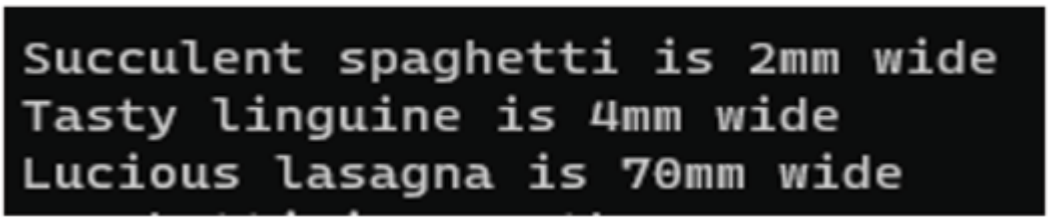

Figure 3.3: Output from TasteThePasta

This also shows that the asserts still worked; saying we implemented the equality_comparable and totally_ordered concepts, even though we did not do so explicitly.

Let us add another member variable to one of our classes:

```
{


: public Chapter3_6_1Food::Pasta {
```

```
Spaghetti() : { }

=

};

};
```

In our function, we will leave everything as it was, but create a couple of instances of spaghetti and compare them.

```
{



{

<<< <<< __FUNCTION__ <<<



=

pastaCount> pastas

{ Chapter3_6_1Food::Linguine{}, Chapter3_6_1Food::Lasagna{},
Chapter3_6_2Food::Spaghetti{} };

pastas.end());
```

```
for p : pastas) {


p.ShowInfo();

  }

<<

  cooked5minutes;

  cooked5minutes.IsAlDente =

  cooked8minutes;

if (cooked5minutes == cooked8minutes) {

<< "spaghetti is the

  }

else {

<< "spaghetti is NOT the

  }
```

<<

}

};

For the most part there is no surprise in the output, as can be seen in the Figure

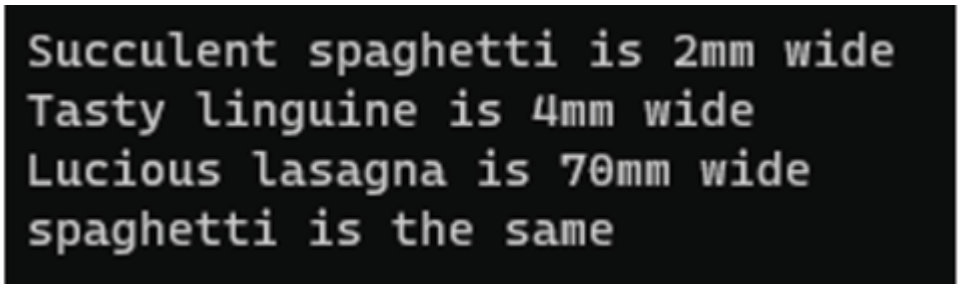

Figure 3.4: Output from revised TasteThePasta

The sort still works, and when we compare our spaghetti objects, it says they are the same, despite one being cooked al dente (cooked, but still firm). Although not preferable, it does make sense, because we are only using the base Pasta comparison operator, which does not know anything about the IsAlDente flag.

If we go back to our definition and just the three-way and equality operators looking at our flag, as shown:

{

```
: public Chapter3_6_1Food::Pasta {


Spaghetti() : { }

=

Spaghetti& other) const {

<=> other.IsAlDente;

  }

== Spaghetti& rhs) const {

== rhs.IsAlDente;

  }

};

};

{
```

```
{

<< << << __FUNCTION__ <<



=



pastaCount> pastas

{ Chapter3_6_1Food::Linguine{}, Chapter3_6_1Food::Lasagna{},
Chapter3_6_3Food::Spaghetti{} };

pastas.end());

for p : pastas) {

p.ShowInfo();

  }

<<

  cooked5minutes;
```

```
  cooked5minutes.IsAlDente =

  cooked8minutes;

if (cooked5minutes == cooked8minutes) {

<< "spaghetti is the

  }

else {

<< "spaghetti is NOT the

  }

<<

}

};
```

Then our output shows items are still sorted, and it can tell how well done our spaghetti is cooked. Refer to Figure

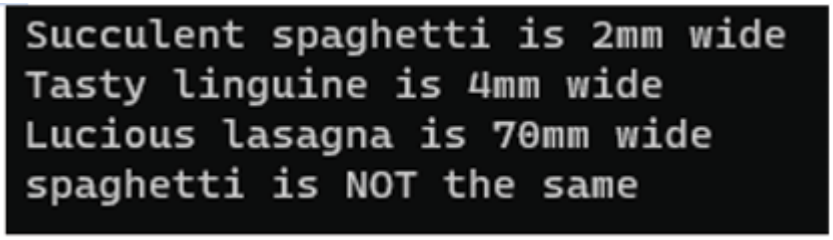

Figure 3.5: Output after we modified the class

This works as we wanted in our case, but we are still implementing whatever operators we need. In C++ 20, they reworked two operators, starting with == and so that they can be defaulted. This can be used if you only have one member variable, or if they are defined in order or importance. In our Pasta scenario, we also had the name member variable, which we did not want to be factored into our ordering. Hence, we defined our three-way operator explicitly.

We will now create a new class, that has a couple of member variables in it. You will notice that for the three-way comparison operator, we are setting it to default. What that means is that the compiler will take each member variable in sequence and compare them. If we have objects ci1 and it will check ci1.Temperature vs and then if they match it will check ci1.Minutes vs This allows you, with very little code, to have a detailed comparison, as follows:

{

{

```
temp, int time) : Temperature(temp), Minutes(time) { }

CookingInstructions& other) const =

};



item1, result, item2) {

if (result == {

<< item1 << " is less than " << item2 <<

  }

(result == {

<< item1 << " is more than " << item2 <<

  }

(result == {
```

```
<< item1 << " is the same as " << item2 <<
  }
}


{
<< << << __FUNCTION__ <<




  CookingInstructions howToCookChicken.Minutes);
howToCookChicken <=> howToCookGrilledCheese, "grilled
howToCookGrilledCheese <=> howToCookChicken2,
howToCookChicken <=> howToCookChicken2, "other
```

```
}

};
```

Note that just by using the default implementation of one operator, the compiler says we have satisfied the constraints of 3 different concepts. The output can be seen in Figure

```
chicken is less than grilled cheese
grilled cheese is more than chicken
chicken is the same as other chicken
```

Figure 3.6: Output from CheckTheOven

Note: Having the default implementation is great in terms of less code, but having a default makes the code prone to other unforeseen issues.

If a developer was looking at the code, and inadvertently switched the order of the variables, as shown:

```
{

{
```

```
int Minutes; // oops, switched the property order


temp, int time) { Temperature = temp; Minutes = time; }

CookingInstructions& other) const =

};


item1, result, item2) {

if (result == {

<< item1 << " is less than " << item2 <<

  }

(result == {

<< item1 << " is more than " << item2 <<

  }

(result == {
```

```
<< item1 << " is the same as " << item2 <<


  }

}



{

<< << << __FUNCTION__ <<









  CookingInstructions howToCookChicken.Minutes);

howToCookChicken <=> howToCookGrilledCheese, "grilled

howToCookGrilledCheese <=> howToCookChicken2,

howToCookChicken <=> howToCookChicken2, "other
```

```
}

};
```

Then they would have changed the sort order and comparison behavior, as can be seen in Figure

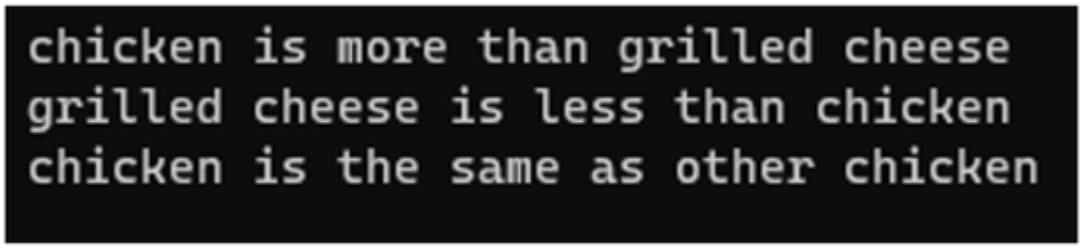

Figure 3.7: Output from modified class

The constructor, concepts, asserts, and other code still work the same, but the comparison result be very different.

Suggestion

That is why we suggest being very careful with defaulting the three-way comparison operator, and whenever possible, be explicit in how you expect objects to be ordered. Whichever way the variables are ordered, it was not clear what is the best sort order. It is better to have that in explicit sort methods, so that it is clear in the code, which can then be tested, and the code-consumer has options on how they want a collection sorted.

Recipe 3.7: three_way_comparable_with

Problem: It would be good to have an easy comparison operator between different types.

Solution: Following a similar pattern, C++ 20 has a three_way_comparable_with concept.

To round out our discussion about comparison concepts, we will include a mention of this. You can use this concept just as equality_comparable_with and to check if there is a three-way comparison operator between T1 and T2 that would relate to a common type TC.

Your utility code might look like:

```
{



template T, U>

T& lhs, const U& rhs) {

U>) {

<=> rhs;
```

```
  }
else {
if == rhs) {

  }
< rhs) {

  }
else {


  }
  }
}
{
```

```
value;

};

{



OurThing& other) const {

if = Id <=> other.Id; cmp != {


  }

return other.ExternalValue.value);

  }

{

+ "+" + ExternalValue.value;

  }

};
```

```
{

<< << << __FUNCTION__ <<

OurThing 3 };

OurThing 3 };



OurThing 5 };

auto result23 = ot2 <=> ot3;

if (result23 == {

<< ot2.ToString() + "==" + ot3.ToString() <<

  }

(result23 == {

<< ot2.ToString() + " > " + ot3.ToString() <<

  }

else {

<< ot2.ToString() + " < " + ot3.ToString() <<
```

```
  }

auto result12 = ot1 <=> ot2;

if (result12 == {

<< ot1.ToString() + "==" + ot2.ToString() <<

  }

(result12 == {

<< ot1.ToString() + " > " + ot2.ToString() <<

  }

else {

<< ot1.ToString() + " < " + ot2.ToString() <<

  }

}


};
```

This could be a handy way to get the same result, even if you use classes that do not use that operator.

We get the expected output, as seen in Figure using our mechanism. It can be used for any type, so it makes more sense to put it into a utility class.

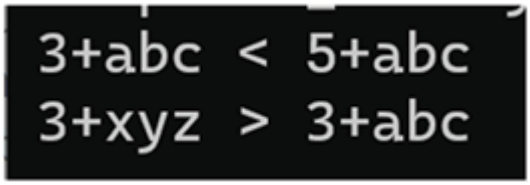

Figure 3.8: Output from TryToCompare

Conclusion

We did not have a long list of concepts to delve into, but we still covered a lot of ground. The comparison templates are a mechanism in knowing the abilities of the type you might be working with. Understanding the limits of a type is good knowledge in many ways, and certainly when you want to compare, or sort objects – it is important to understand what is possible. Concepts continue to be a help to your code; whether you just use static assertions, or create functions that only accept a type that conforms to a particular concept, or whether you create a new concept that require some existing concept constraints.

In the next chapter, we will continue with another area of concepts – those relating to iterators.

Points to remember

Several new C++20 concepts were created to help understand what comparison capabilities/constraints your type has.

A new header also includes utility classes and functions for looking at the comparison result.

There are new default == and <=> operators, that can help cut out a lot of code.

References

StackOverflow. (n.d.). Retrieved from https://stackoverflow.com/questions/66937947/why-is-unique-ptr-not-equality-comparable-with-nullptr-t-in-c20

ISO CPP. Retrieved from http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Ro-symmetric

Walletfox. (n.d.). Retrieved from https://www.walletfox.com/course/sortvectorofcustomobjects.php

CppCon. (n.d.). Retrieved from https://www.youtube.com/watch?v=8jNXy3K2Wpk

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 4

Using the New Iterator Concepts

Introduction

In the previous chapter, we learned that something that sounds simple such as comparison, has a lot of different pieces to it. For this chapter on iterator concepts, by the time we are done, you might get the same feeling. Unlike other concepts, some of these iterator concepts are a piece of the puzzle. We are not saying it is complicated, but there are different things to consider.

You can think of an iterator just as a pointer that walks through a collection, and you can look at the de-referenced value, meaning the value that the iterator is pointing to. Of course, there are different ways to create or use an iterator. Iterators can be just for reading, or just for writing (and of course, both). Iterators can start at the beginning, the end, or somewhere in the middle of your collection. We will go into the different concepts available, so that you can decide which is the best concept to use when creating function arguments, or when creating your own concepts. This chapter will also look at sentinels, and Unlike previous chapters, you probably will not reuse most of these concepts. Many of these smaller iterator concepts are to describe facets of what the larger concepts do. They are still worth mentioning, because we will see them as requirements for concepts in future chapters.

Structure

In this chapter, we will cover the following recipes for the concepts:

Recipe 4.1: weakly_incrementable

Recipe 4.2: input_or_output_iterator

Recipe 4.3: sentinel_for

Recipe 4.4: sized_sentinel_for

Recipe 4.5: indirectly_readable

Recipe 4.6: indirectly_writable

Recipe 4.7: incrementable

Recipe 4.8: input_iterator

Recipe 4.9: output_iterator

Recipe 4.10: forward_iterator

Recipe 4.11: bidirectional_iterator

Recipe 4.12: random_access_iterator

Recipe 4.13: contiguous_iterator

Recipe 4.14: indirectly_movable, and std::indirectly_movable_storable

Recipe 4.15: indirectly_copyable, and std::indirectly_copyable_storable

Recipe 4.16: indirectly_swappable

Recipe 4.17: indirectly_comparable

Recipe 4.18: permutable

Recipe 4.19: mergeable

These concepts can get rather detailed, which is one reason why we did not go into too many samples, but on the bright side – they should all work if you are using one of the three main compilers:

```
{

// works with compiler?            when tested

Chapter4_1::WeaklyIterateThroughIngredients(); //  2023.11.04  yes  yes  yes
```

```
Chapter4_2::IterateAndUpdate();          //2023.11.04 yes

Chapter4_10::LookAtCollectionsForward(); //2023.11.04 yes

yes yes yes

Chapter4_12_1::SimpleLookAtVector();    //2023.11.04 yes

Chapter4_12_2::SimpleLookAtVector();    //2023.11.04 yes

yes yes yes

}
```

Objectives

In this chapter, you will learn about the new concepts, and their differences, including the why and how to use them. We will see that these concepts can be powerful in making sure the code conforms to the requirements of the concept. Not only will these concepts be helpful with your code, but they will also prepare you for when we get into ranges later on in this book.

Recipe 4.1: weakly_incrementable

Problem: We want an easy way to incrementally go through our collection.

Solution: C++ 20 has a new concept named

This concept is more of a small step into the subject; it might not be an end goal, but a step along the journey. The weakly_incrementable concept means that you can increase the iterator, but the iterator is not assured to abide by which we learned about earlier. In other words, we can perform it++ (or but that is about it. If we had two iterators, it1 and and even if it1 == it does not guarantee that ++it1 ==

The other point about this concept is that for weakly_incrementable types, they should be single-pass, meaning that you are not able to iterate through the collection multiple times with the same iterator.

Two other concepts to mention quickly: one being weakly_incrementable concept requires the concept std::movable that we will talk more about in future chapters. So, we know it can be move constructed and moved assigned. The other concept it requires is which has a longer definition. However, for now, we will say that the iterator will hold a reference to something that is not just void. It does not make sense, nor possible, to have a collection of voids, so of course one is not able to iterator them.

To walk through a quick example of what we mean, we shall first list the header files we will be using throughout this chapter:

#include

#include

#include

#include

#include

#include

#include

#include

#include

#include

#include

Here is a simple example:

template T>

T::iterator>

```
collection) {

= collection.begin();

auto count =

while collection.end()) {

<< "Adding ingredient: " <<

if (count % 2 == {

<<

iter++;

  }

else {

<<



  }

<<
```

```
 count++;

}

}

{

{

<< << << __FUNCTION__ <<

5> ingredientsToAdd = { "hot "course "pepper" };

CycleThroughIngredientsPreAndPost(ingredientsToAdd);

}

};
```

The results show in Figure that we have iterated through our list of ingredients. We alternated with every other to pre and post incrementing the iterator.

```
Chapter4_1::WeaklyIterateThroughIngredients
Adding ingredient: hot sauce                    (post-increment)
Adding ingredient: oregano                      (pre-increment)
Adding ingredient: vinegar                      (post-increment)
Adding ingredient: course salt                  (pre-increment)
Adding ingredient: pepper                       (post-increment)
```

Figure 4.1: Output from WeaklyIterateThroughIngredients

Recipe 4.2: input_or_output_iterator

Problem: We need a way to reference the value via our iterator.

Solution: Released with C++ 20, we have the concept

Our next step with iterator concepts relates to the ability to get at the value our iterator is looking at. With the previous weakly_incrementable and the current focus of we have the basis for all of our other iterator concepts. We still cannot do everything we want:

We can increment, but we are not sure how to tell when to stop incrementing.

We cannot always read and write values in all cases.

We still do not have all the capabilities we will need, such as going in either direction, or random access.

A quick answer of what std:input_or_output_iterator does, is that it allows you to dereference, or get at the value the iterator is looking at. It seems a bit unrealistic to think that you can iterate through a collection of values without being able to see any of the values. These initial concepts though, are looking at the basics, and that is why we have this one explicitly focused on retrieving the value.

In this example, we will iterate through a set of numbers. The templated function looks at a std:input_or_output_iterator collection, allowing it to update the value:

```
template T>

T::iterator>


collection) {

= collection.begin();

auto count =

while collection.end()) {

if (count % 2 == {

<< "(changing value iterator is looking

auto temp =

  temp++;

  = temp;

  }

<< "value=" << <<

  count++;
```

```
iter++;

}

}

{


title, numbers) {

<< << "Printing numbers " << title <<

for it = numbers.begin(); it != numbers.end(); it++) {

if != numbers.begin()) {

<< ",

  }

<< *it;

  }


<< <<
```

```
}


{

<<< <<< <<< __FUNCTION__ <<<

numbers= { 77 };

numbers);

CycleThroughNumbersToUpdate(numbers);

numbers);

}

};
```

Looking at the results in Figure we have changed the value of every other item in the collection that we iterated through:

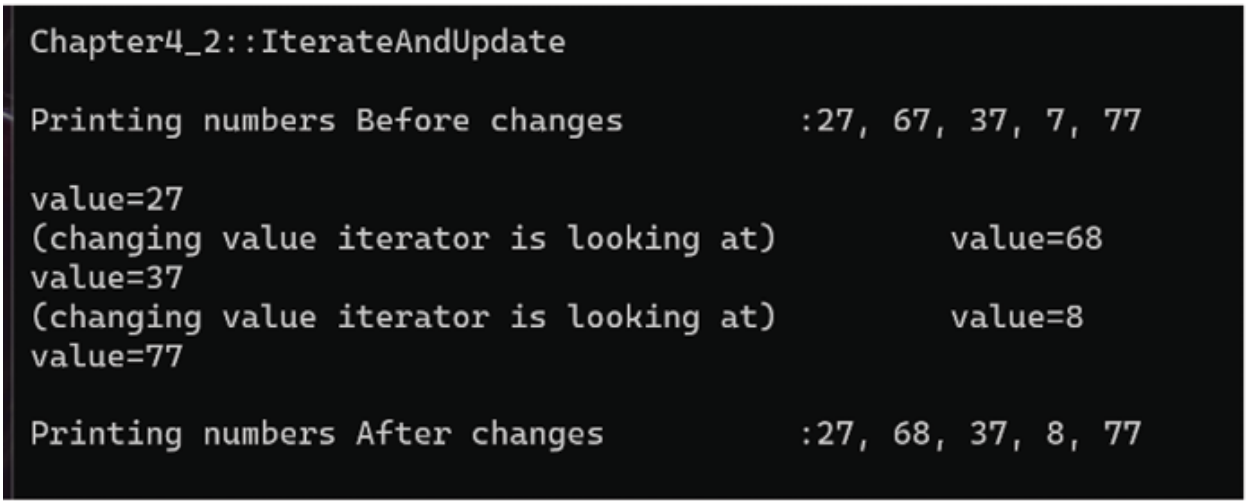
Chapter4_2::IterateAndUpdate
Printing numbers Before changes :27, 67, 37, 7, 77
value=27
(changing value iterator is looking at) value=68
value=37
(changing value iterator is looking at) value=8
value=77
Printing numbers After changes :27, 68, 37, 8, 77

Figure 4.2: Output from IterateAndUpdate

Recipe 4.3: sentinel_for

Problem: We need a way of knowing when our collection ends.

Solution: Also present in C++ 20, the concept is

This next concept sounds like another baby step, but it is something we can go into a bit more detail about. It makes sense that after we know how to increase an iterator, we need to know when we should stop iterating. An important part of an iterator is to know when you get to the sentinel value. You can think of simple cases such as:

When looking at a null-terminated string, you know you are done when you find the null character.

When looking at a null-terminated list, if you get to the null value then you know you are at the end.

When looking through a collection for a particular value, some functions return a -1 if they cannot find the value.

A sentinel value can be a different type, other than the type that is in the collection you are looking at. The latter example also shows that the sentinel may not be a value in your collection. This means that you do not always check if your iterator is looking for a sentinel, but whether your sentinel function says you are now done iterating. The template concept sentinel_for has two type parameters: One for the sentinel and one for the

iterator, indicating the sentinel itself can always tell if it is done without having the iterator.

The nomenclature of a range is [i, meaning, that for an iterator of type i and a sentinel of type we will have zero or more values in the range of the type for which the iterator is looking at.

Since we have not used much of a recipe analogy while mentioning these last few concepts, we can mention one now. When following a recipe on a page, you will have a list of ingredients and a list of steps.

Let the ingredient list be as follows:

Chicken breast

Spice mix

Fruit glaze

The steps are as follows:

Wash chicken, and dry off completely.

Coat chicken with spice mix.

Put chicken in pre-heated oven for 30 minutes, or until golden brown.

Once cooked, remove from oven and let it rest for 5 minutes.

Warm up the glaze for 1 minute.

Coat with glaze.

You may not know how many things are on the list, which means you will not know where the end of the list is until you get there. Meaning when following a recipe, you will have a bunch of steps, and once you finish the last step in the list of instructions will you realize that you have completed the recipe. Or maybe there are multiple sets of instructions; one list to perform steps before you put the dish into the oven, and another list of steps to do once it comes out of the oven fully cooked. Although they are all steps in the recipe, you understand that there are different sections, or groups of instructions. You could say that for each group of instructions, there is a sentinel to denote that this is the last instruction for this group.

Thinking about when your chicken is cooking in the oven, you may not use the cooking time as if it was set in stone. The recipe will most likely say estimated cooking time meaning it is not an exact amount of time for the chicken to be ready. If it does not seem done, and you give it another two minutes, and then another 30 seconds; you are checking the condition IsChickenCooked() function rather than relying on the number of steps. The process of knowing when a chicken is cooked is different from the process of heating the glaze. You can understand when you have completed both steps, but how you evaluate when you are done, or in more technical jargon, when you get to the done are measured in different ways.

We are still not at the place where we can do everything with iterators that we want to do. It is just good to know why they split these concepts into tiny pieces because different concepts can be used in different ways later on.

One other note to add, that will be mentioned a few different times. There is no clear linear way to explain all of the different concepts, without jumping back and forth. Some concepts relate to others that have not been discussed yet. A minor point, but std::sentinel_for depends on but we have not talked about that yet. For now, we will say that means the type needs to be copyable and have a default constructor, but we will talk about it more in the upcoming chapter. We have also mentioned that you may not use the concepts much by themselves, but they will become more important later, once we talk about ranges.

Recipe 4.4: sized_sentinel_for

Problem: There are cases where we need to know how many items the iterator has to go through to get to the sentinel.

Solution: C++ 20 has a new concept of

We will not spend a lot of time on this concept, since it seems to be another building block in a collection of iterator concepts, but it is mostly saying that the time to make each iterator is constant.

If we have an iterator of type and a sentinel of type we will have a range of those instances denoted as [i, s).

Also existing is another type, named meaning the delta, and so we would have n iterations before i gets to

Our concept sized_sentinel_for states that the distance should be in constant time, and so if we have a range of [i1, there would be n1 iterations, and if our range was twice as big [i2,s) then n2 is twice as big as

Curiously, it does not mention the range being contiguous when defining this concept, but we will look at the concept contiguous_iterator later in this chapter.

Recipe 4.5: indirectly_readable

Problem: We want a concept about reading a dereferenced value, that preserves equality.

Solution: The next concept we will talk about is

This concept is interesting, in that it does not require any of the previous concepts that have so far been mentioned in this chapter. It does not progress on the idea of an iterator but focuses on being able to get at the value but using the * operator. It can be used with iterators, and with pointers. Aside from the ability to read the value, the other essential part of this concept is the preservation of equality.

A brief description is that it needs to meet the requirement, if you have a const pointer, and a non-const pointer, dereferencing both would give you the same type.

The other entity to mention is called This type computes the associated value, and so in some cases, you might use this, but most would use especially when dealing with an iterator.

There is also a which relates to the properties of the iterator, or mostly LegacyIterator types.

Recipe 4.6: indirectly_writable

Problem: We want a concept about writing a dereferenced value, that preserves equality.

Solution: Similar to there is also

This counter side to the previous concept, of course, deals with the ability to set, or write the value to a dereferenceable object. Instead of the * operator being used on the right-hand side, indirectly_writable has the * operator being used on the left-hand side. Of course, it too needs to have equality preservation.

We can also refer back to our std:input_or_output_iterator concept. The ability to read a dereferenced value would relate to while the ability to update a dereferenced value would relate to

Recipe 4.7: incrementable

Problem: We want a concept that is incrementable, but also preserves equality.

Solution: The next progression after weakly_incrementable is

Unsurprisingly, our concepts are getting bigger and more advanced. The concept incrementable uses weakly_incrementable that we learned from earlier, as well as We know it ensures that it is: copyable, default constructable, and equality comparable. Now, we have a concept that can do most of what we require.

Another entity to briefly mention is which relates to the different types of the iterator. Again, it might not be something that you use, but when creating other concepts, or other functions that have parameters that you could denote as a concept, then it is good to know what exists. That way, you can use an existing concept, and that will ensure you fulfill all the requirements of the concept, or the consumer of your code uses the appropriate type that meets the requirements of your concept. Either way, it should mean safer, cleaner code.

Recipe 4.8: input_iterator

Problem: We need a concept that is like the basic input_or_output concept, but can also read values.

Solution: In C++ 20, they also defined

Moving on, the input_iterator combines input_or_out_iterator and and so now, our concept can read the dereferenced value.

Recipe 4.9: output_iterator

Problem: Similar to the requirements of we need a concept that can indirectly write a value.

Solution: Included with all of the C++ 20 concepts is

The next concept to speak allows us to iterate and write the value. The template has two type parameters: I and T. For equality_compariable is not required.

Recipe 4.10: forward_iterator

Problem: We want a reading iterator that can do multiple passes.

Solution: We will look at C++ 20's

Progressing nicely through our iterator journey, we have something more than just a little piece of the puzzle. The forward_iterator concept combines other concepts discussed in this chapter: The and our recent Now, we have something that can do multiple passes, and read the value of the iterator.

Since we now have enough information, we can show a worthwhile code sample as follows:

```
template < typename T>

T::iterator>

title, T collection) {

= collection.begin();

= collection.end();

= startingIter;
```

```
<< "first pass of : " << title <<

for (; curentIter endIter; ++curentIter) {

if startingIter) {

<< ",

  }

<<

}

<<

//

<< "second pass:" <<


for (curentIter = startingIter; curentIter != endIter; ++curentIter) {

if startingIter) {

<< ",
```

```
  }

<<

}

<< <<

}

{


{

<< << << __FUNCTION__ <<

vectorNumbers = { };

of vectorNumbers);

dequeNumbers = { };

queue of dequeNumbers);

5> arrayLetters = { };
```

```
of arrayLetters);

set = { };


forward_list = { };




// does not compile

// CycleThroughItems("queue", queueNumbers);

}

};
```

The output can be seen in the following Figure

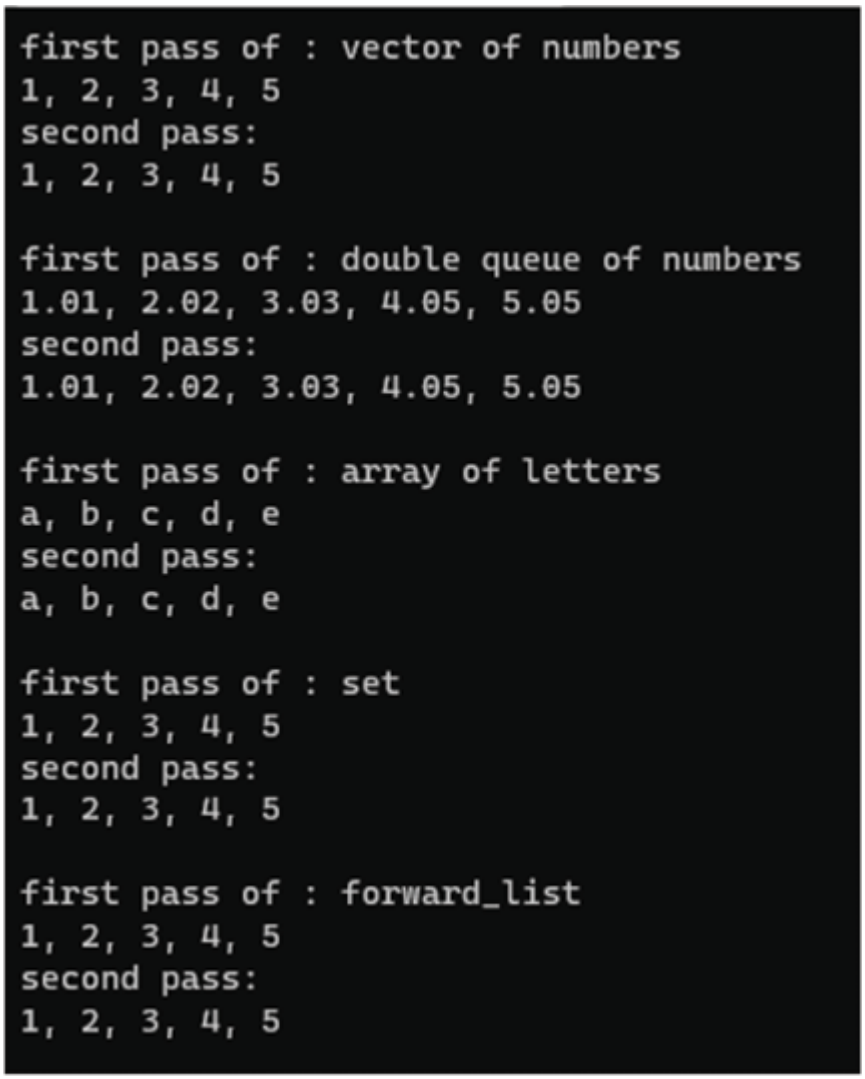
first pass of : vector of numbers
1, 2, 3, 4, 5
second pass:
1, 2, 3, 4, 5
first pass of : double queue of numbers
1.01, 2.02, 3.03, 4.05, 5.05
second pass:
1.01, 2.02, 3.03, 4.05, 5.05
first pass of : array of letters
a, b, c, d, e
second pass:
a, b, c, d, e
first pass of : set
1, 2, 3, 4, 5
second pass:
1, 2, 3, 4, 5
first pass of : forward_list
1, 2, 3, 4, 5
second pass:
1, 2, 3, 4, 5

Figure 4.3: Output from LookAtCollectionsForward

We now see that we can iterate through a number of different types of collections, of various types. We did not need to specify what it was (although of course, the auto declaration took care of a lot of that). Moreover, just to make sure we could do multiple passes, we looped through twice.

We created a template type that requires the so that we can still pass in the collection. This is different from how we pass an auto parameter, as we have done in recent chapters.

Note with our last collection, that the queue did not meet the requirements of a You are not able to iterate a queue. You can only push a value into the end of a collection, and pop the first value from the collection. We are not able to iterate to see all the values that exist in the collection.

Recipe 4.11: bidirectional_iterator

Problem: Since we have an iterator that can go forward, we also need one that can go backwards.

Solution: Also present in C++ 20, the concept is

We made good progress with which is a requirement for That, plus defining that you can decrement the iterator instance, whether that is or

We can build off of our code sample we had before:

```
template < typename T>

T::iterator>

title, T collection) {

= collection.begin();

= lastIter--;

= lastIter;

<< "first pass of : " << title <<
```

```
for ; {

if lastIter) {

<< ",

  }

<<

if == startingIter)

}

<<

<< "second pass:" <<

for = lastIter; ; {

if lastIter) {


<< ",

  }
```

```
<<

if == startingIter)

}

<< <<

}

{


{

<< << << __FUNCTION__ <<

vectorNumbers = { };

of vectorNumbers);

dequeNumbers = { };

queue of dequeNumbers);

5> arrayLetters = { };
```

```
of arrayLetters);

set = { };


forward_list = { };



}


};
```

The main tweak we had to do was decrement the end iterator. Since that would point to something outside of the range we wanted, decrementing it would bring it to the valid last item. The other thing was to tweak the logic so that we break out of the loop after we output the first element. Normally, people might use rbegin and but then we would not be decrementing the iterator.

The output can be seen in the following Figure

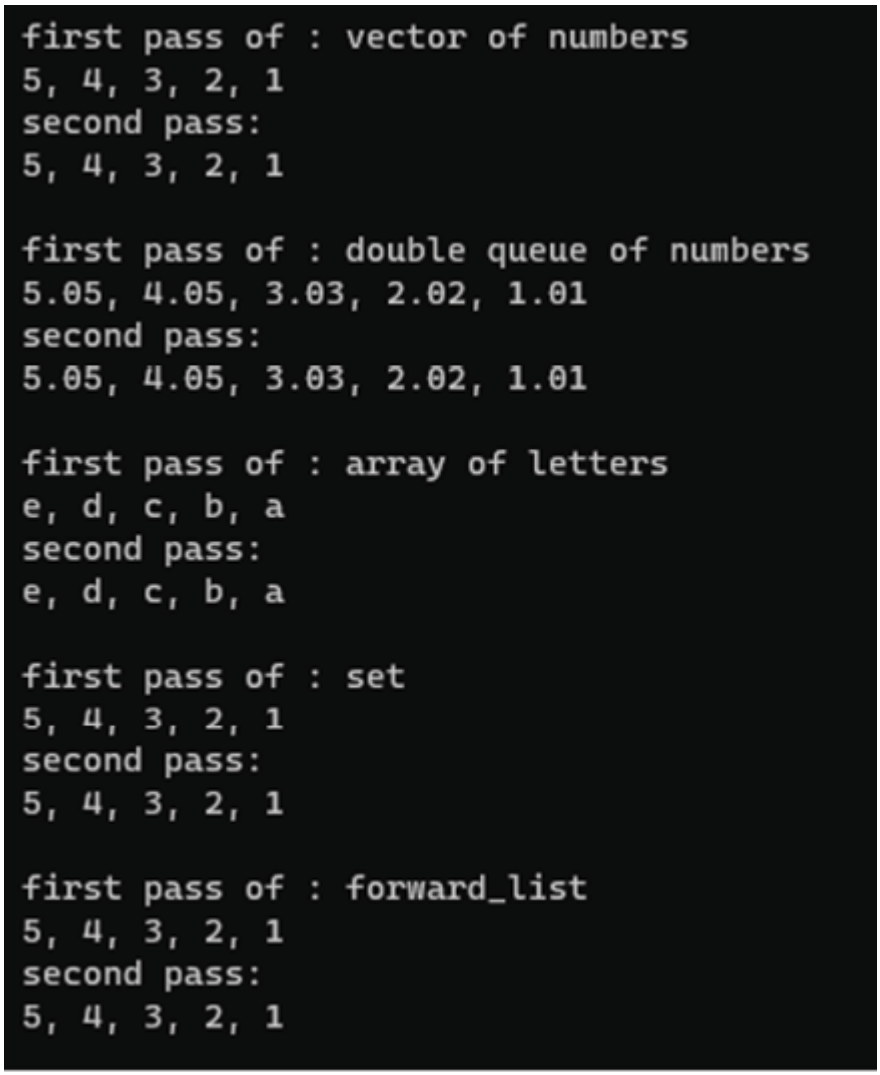
first pass of : vector of numbers
5, 4, 3, 2, 1
second pass:
5, 4, 3, 2, 1
first pass of : double queue of numbers
5.05, 4.05, 3.03, 2.02, 1.01
second pass:
5.05, 4.05, 3.03, 2.02, 1.01
first pass of : array of letters
e, d, c, b, a
second pass:
e, d, c, b, a
first pass of : set
5, 4, 3, 2, 1
second pass:
5, 4, 3, 2, 1
first pass of : forward_list
5, 4, 3, 2, 1
second pass:
5, 4, 3, 2, 1

Figure 4.4: Output from LookAtCollectionsBackward

We see our same values outputted backwards, twice.

Recipe 4.12: random_access_iterator

Problem: In some cases, we need to be able to position the iterator as a specific item, and not just increment or decrement to get a new position.

Solution: We can use

The definition for the concept random_access_iterator is now becoming quite complex. We use which gives us a lot, such as the following (assuming col is the collection, iter is the iterator, and n is some integer):

iter++, or ++iter

iter—, or --iter

However, we also have the operations:

iter += n

iter -= n

col[n]

When looking at some sample code:

```
template < typename T>

T::iterator>

collection) {

= collection.begin();

<< "current value at beginning=" << *iter <<

iter++;


<< "current value after inc 1 =" << *iter <<

iter--;

<< "current value after dec 1 =" << *iter <<


<< "current value after inc 1 =" << *iter <<


<< "current value after dec 1 =" << *iter <<

//
```

```
iter +=

<< "current value after inc 3 =" << *iter <<

iter -=

<< "current value after dec 2 =" << *iter <<

//

<< "current value of fourth # ="

  << <<

}

{


{

<< << << __FUNCTION__ <<

vectorNumbers = { };

FirstSimpleLookAtRandom(vectorNumbers);

}
```

```
};
```

We can see that we can not only increment or decrement multiple places but also be able to set it to an absolute location [n] is extremely useful.

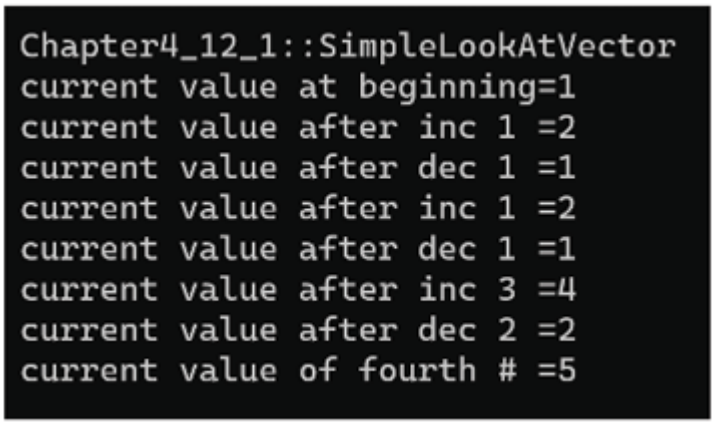
Chapter4_12_1::SimpleLookAtVector
current value at beginning=1
current value after inc 1 =2
current value after dec 1 =1
current value after inc 1 =2
current value after dec 1 =1
current value after inc 3 =4
current value after dec 2 =2
current value of fourth # =5

Figure 4.5: Output from Chapter4_12_1:: SimpleLookAtVector

One other entity that is sometimes overlooked is It has been around for a long time (included in C++98, and then tweaked a bit in C++17). It can tell you the distance between two iterator points, it can even be between begin() and Let us create another example that has the random_access_iterator constraint. We will create a method called this will accept a parameter telling it to move the iterator backwards or forwards the given number of places. To make it simpler, we have not done any bounds checking, so if the iterator was on the third element and we told the iterator to move -2, it would then be on the first item. If sitting on the third element and told the iterator to move -5, then we would get an

exception. We will use it to show that we can satisfy the std::random_access_iterator by moving multiple places.

```
{


auto beginning,

int delta,

current) {


if (delta == {


  }

(delta == {


  }

(delta < {

  current -=
```

```
  }
else {
  current += delta;
  }
//
auto distanceFromBeginning = current);
<< "after moving " << delta << ", value@"
  << " [" << distanceFromBeginning << "]=" << *current

  << ", distance changed=" << current)
  <<
}
T>
T::iterator>
```

```
collection) {

// start with the current at the beginning

auto current = collection.begin();

RepositionIterator(collection.begin(), current);

RepositionIterator(collection.begin(), current);

RepositionIterator(collection.begin(), current);

RepositionIterator(collection.begin(), current);

RepositionIterator(collection.begin(), current);

<< "current value of fourth # =" << <<

}


{

<< << << __FUNCTION__ <<

vectorNumbers = { };
```

```
<< "values :

auto index =

for it = vectorNumbers.begin(); it != vectorNumbers.end(); ++it) {

if != vectorNumbers.begin()) {

<< ",


  }

<< "v[" << index++ << "]= " << *it;

  }

<< <<

SimpleLookAtRandomDistance(vectorNumbers);


// does not compile since queue does not comply to concept

// SimpleLookAtRandomDistance(q1);
```

```
}

};
```

We are always passing in the begin() value, just so we can calculate the absolute offset. We are also passing in the current iterator, so we can calculate the distance after we reposition our iterator. The extra logic for performing the repositioning was just to verify that iter += and iter -=n still work fine. Showing the distance changed always being the same as our planned delta, proves we can use a random_access_iterator with this container. The output is shown in Figure

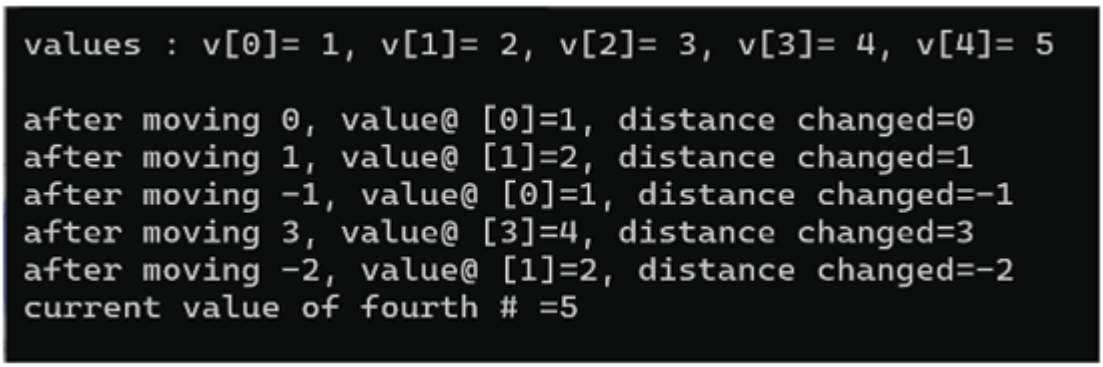
values : v[0]= 1, v[1]= 2, v[2]= 3, v[3]= 4, v[4]= 5
after moving 0, value@ [0]=1, distance changed=0
after moving 1, value@ [1]=2, distance changed=1
after moving -1, value@ [0]=1, distance changed=-1
after moving 3, value@ [3]=4, distance changed=3
after moving -2, value@ [1]=2, distance changed=-2
current value of fourth # =5

Figure 4.6: Output from Chapter4_12_2::SimpleLookAtRandom

The code shows once again, that std::queue does not comply to the concept, so we are not able to use that type of container.

Recipe 4.13: contiguous_iterator

Problem: We want a concept that ensures us that our elements are stored in contiguous memory.

Solution: C++ 20 provides us with

The contiguous_iterator does not provide us any extra functionality in the concept, but more of a refinement on random_access_iterator that our elements are stored in contiguous memory. Moreover, remember that it does not guarantee contiguity just by saying that your concept or templated class requires it. It is simply a guide on what you need to do in your code, to be able to comply with the concept.

A std::vector and a std::deque both meet the requirements of the random_access_iterator concept, but the deque does not comply with

If we reuse some of our previous code sample:

```
{



auto beginning,

int delta, current) {
```

```
  current += delta;

auto distanceFromBeginning = current);

<< "after moving " << delta << ", value@"


  << " [" << distanceFromBeginning << "]=" << *current

  << ", distance changed=" << current)

  <<

}

T>

T::iterator>

collection) {

// start with the current at the beginning

auto current = collection.begin();
```

```
RepositionIterator(collection.begin(), current);

RepositionIterator(collection.begin(), current);

RepositionIterator(collection.begin(), current);

RepositionIterator(collection.begin(), current);

RepositionIterator(collection.begin(), current);

<< "current value of fourth # =" << <<

}

auto beginning,

int delta, current) {


  current += delta;

//

= current);

=

=
```

```
<< "value after moving " << delta << ", current value@"
  << " [" << absDist << "]=" << *current
  << ", distance changed=" << current)
  << ",\n\t\taddrBefore=" << addrBefore
  << ", addrAfter=" << addrAfter
  << ", address diff=" << - addrBefore)
  <<
}
T>
T::iterator>
collection) {
// start with the current at the beginning
auto current = collection.begin();
```

```
RepositionContiguousIterator(collection.begin(), current);

RepositionContiguousIterator(collection.begin(), current);

RepositionContiguousIterator(collection.begin(), current);

RepositionContiguousIterator(collection.begin(), current);

RepositionContiguousIterator(collection.begin(), current);

<< "current value of fourth # =" << <<

}

{

<< << << __FUNCTION__ <<

dequeNumbers = { };

<< "Look at random

<< "double values :
```

```
auto index =

for it = dequeNumbers.begin(); it != dequeNumbers.end(); ++it) {

if != dequeNumbers.begin()) {

<< ",

}

<< "v[" << index++ << "]= " << *it;

  }

<< <<

SimpleLookAtRandomDistance(dequeNumbers);

vectorNumbers = { };

<< "\ninteger values :

  index =

for it = vectorNumbers.begin(); it != vectorNumbers.end(); ++it) {

if != vectorNumbers.begin()) {
```

```
<< ",

  }

<< "v[" << index++ << "]= " << *it;

  }


<< <<

<< "Look at contiguous

SimpleLookAtContiguous(vectorNumbers);

// fails to compile since a deque is not contiguous

// std::cout << «Look at contiguous deque\n";

// SimpleLookAtContiguous(dequeNumbers);

}

};
```

We have already seen that a vector can be passed to a function that wants a random_access_iterator and we first show that we can pass a deque to the same function as well. The new code has a function that does similar

repositioning of the current iterator, but the function also shows the memory addresses of where the before and after iterators are pointing to. We can pass our familiar vector to the function that wants a but when we try to pass in the it will not compile.

Note that we are using std::to_address also released with C++ 20), which is called a fancy pointer overload. It can be used in some cases where std::addressof(*p) will not work.

Our output can be seen in Figure

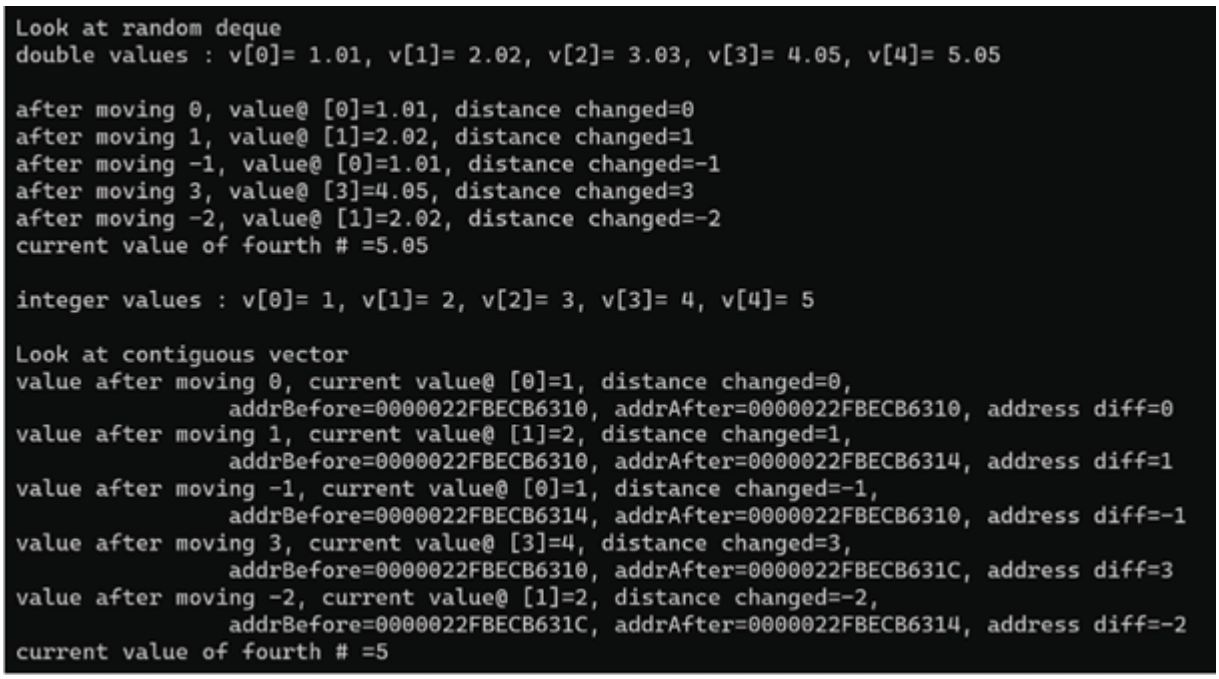
Look at random deque
double values : v[0]= 1.01, v[1]= 2.02, v[2]= 3.03, v[3]= 4.05, v[4]= 5.05
after moving 0, value@ [0]=1.01, distance changed=0
after moving 1, value@ [1]=2.02, distance changed=1
after moving -1, value@ [0]=1.01, distance changed=-1
after moving 3, value@ [3]=4.05, distance changed=3
after moving -2, value@ [1]=2.02, distance changed=-2
current value of fourth # =5.05
integer values : v[0]= 1, v[1]= 2, v[2]= 3, v[3]= 4, v[4]= 5
Look at contiguous vector
value after moving 0, current value@ [0]=1, distance changed=0,
addrBefore=0000022FBECB6310, addrAfter=0000022FBECB6310, address diff=0
value after moving 1, current value@ [1]=2, distance changed=1,
addrBefore=0000022FBECB6310, addrAfter=0000022FBECB6314, address diff=1
value after moving -1, current value@ [0]=1, distance changed=-1,
addrBefore=0000022FBECB6314, addrAfter=0000022FBECB6310, address diff=-1
value after moving 3, current value@ [3]=4, distance changed=3,
addrBefore=0000022FBECB6310, addrAfter=0000022FBECB631C, address diff=3
value after moving -2, current value@ [1]=2, distance changed=-2,
addrBefore=0000022FBECB631C, addrAfter=0000022FBECB6314, address diff=-2
current value of fourth # =5

Figure 4.7: Output from SimpleLookAtVectorAndDeque

We will not focus on the memory address, but if we look at the difference of the memory addresses, we see that they match the distance the iterator travelled, which also matches the offset we attempted to change the iterator.

Note: There are a few other iterator-related entities; some are worth investigating, but we will not get into in this chapter.

Apart from there is also:

iter_value_t: That computes the value type of T.

iter_reference_t and That computes the reference type of T.

That computes the rvalue reference of T.

That computes the common reference type of T.

The last entities to talk about are the iterator tags. For the most part, each type of iterator has an associated tag type:

input_iterator_tag relates to

forward_iterator_tag relates to and derives from

bidirectional_iterator_tag relates to and derives from

random_access_iterator_tag relates to and derives from

contiguous_iterator_tag relates to and derives from

These base structures are empty, and partially relate to legacy iterators, and can also relate if you are doing something specialized with your collection. You can tell there is certainly a hierarchy in a dependency as shown in Figure

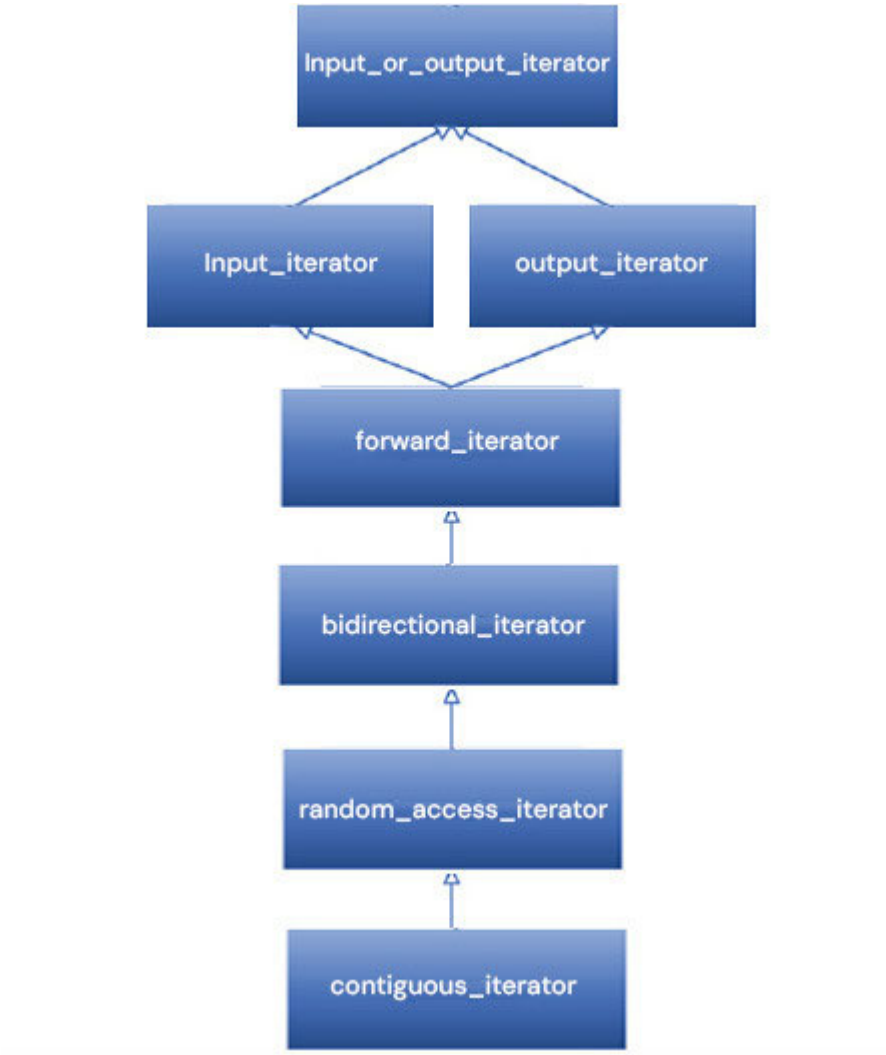

Figure 4.8: View of iterator concept dependencies

Recipe 4.14: indirectly_movable, and indirectly_movable_storable

Problem: We want a concept that combines readability and writability, so we can move a value.

Solution: C++ 20's indirectly_movable has just that.

We will probably go through the next bunch of topics in this chapter fairly quickly. For some of these topics, it is rather a chicken versus the egg question. Some of these concepts certainly deal with the iterator, but they may not be used until we get into ranges later on.

Now that we have concepts that can read and write, we can combine those to create a new concept named indirectly_movable, which can move a value from one address to another.

The concept indirectly_movable_storable combines a few more requirements but has a similar goal. This concept also says that the value can be moved from the indirect type through an intermediate object. This concept probably falls under the category of ones you will not use first-hand, but they are reused in future concepts.

Recipe 4.15: indirectly_copyable, and indirectly_copyable_storable

Problem: We want a concept that combines readability and writability, and so we can copy a value.

Solution: C++ 20 also has

These concepts are also very similar to the indirectly_movable ones. There is a difference in the requirements that allow the movable concepts to move the value, whereas these only copy the value. The indirectly_copy_storable again will probably be something you do not normally use yourself.

Recipe 4.16: indirectly_swappable

Problem: We want a concept where you can read and write values so that they can be swapped.

Solution: C++ 20 has the indirectly_swappable concept.

The concept indirectly_swappable builds off of what we have so far. Instead of thinking that we have an in-reference and an out-reference, this deals with two iterators, saying it can swap the value between them. This is primarily used when dealing with ranges, and it requires ranges::iter_swap to work. Thus, we will certainly get into more details about ranges in a later chapter, but included it in the iterator chapter for now.

Recipe 4.17: indirectly_comparable

Problem: We want a concept that can indirectly compare two values.

Solution: From C++ 20, we have

This seems to be another intermediate concept, that you probably will not use first-hand, but other concepts we look at later will use it. The concept specifies the means to compare values across independent ranges.

Recipe 4.18: permutable

Problem: We want a concept that allows the iterator to move values.

Solution: Nearing the end of our iterator concept list, we have

This concept combines others we recently talked about a forward_iterator that is and Thus, we can iterate through a collection, and when required, call an intermediate object to move a value, or swap two different values.

Recipe 4.19: mergeable

Problem: We want a concept to know when a templated type can be merged together.

Solution: There is introduced in C++ 20.

The concept mergeable is the last on our iterator list (until we get back into ranges). A variation on other recent concepts we have looked at, this allows two different ranges to be merged together into a single output range. It also requires the indirect_strict_weak_order concept, which we will talk about later in our chapter about Callables.

Conclusion

When discussing iterator concepts, it took us quite a while to get into the code. It comprises different features or mechanisms that are built up into something more tangible. You might use just the larger concepts, or you might need a smaller concept for something specific. These concepts can be helpful as function type parameters so that you know your code complies with that type, or safer still, it ensures others using your code, comply. We also created a few templated functions, and that helped us ensure our code would compile when trying different types of collections.

The book will definitely talk more about iterators again later on when discussing ranges, but for now, we will go over the other concepts.

Points to remember

There are several concepts to describe various capabilities/restrictions of an iterator.

You can use an iterator concept as a function parameter, without knowing the exact type of collection, or the type the collection is holding.

There are a number of helper classes and functions that can help enhance your code.

References

To_address. (n.d.). Retrieved from https://en.cppreference.com/w/cpp/memory/to_address

CHAPTER 5

Using the New Object Concepts

Introduction

So far, this book has introduced you to concepts and discussed some of the predefined concepts that were shipped with C++ 20. An earlier chapter mentioned some concepts related to investigating (or restricting, depending upon how you look at it) how a type is constructed. This chapter will continue along the long path of concepts, discussing some of the capabilities of an object. We will go into how you can use and reuse some of these concepts, and how they will benefit your code.

Structure

In this chapter, we will cover the following recipes:

Recipe 5.1: movable

Recipe 5.2: copyable

Recipe 5.3: semiregular

Recipe 5.4: regular

The features in this chapter are fairly fundamental to the language, which contributes to the reasoning that all three major compilers have completed these features.

{

// works with compiler? when tested

 //2023.12.09 yes

//2023.12.09 yes

yes

yes

//2023.12.09 yes

}

Objectives

By the end of this chapter, the reader will learn how to check some of the powerful traits of C++ objects: whether they are copyable, mergeable, or movable. The reader will also get to know what a regular or semiregular object means as well as a review of the rules.

Recipe 5.1: movable

Problem: We need a mechanism to validate whether an object can be moved.

Solution: C++ 20 has a movable concept.

Suggestion

It may be a big diversion, but let us first go back and review the recommendations that exist, and did exist for the constructors and operators that should be coded. In C++ 11, they came out with the move constructor, and that changed a lot of how code should be written. Before C++ 11, if you made a class where you coded:

a destructor

possibly a copy constructor

or maybe a copy assignment operator

Then the Rule of Three states that if you have at least one of those, then you should have all 3 of them.

C++ 11 included the move constructor and move operator. C++ was considered a fast language before, but the ability to have another object take ownership of the allocated memory, instead of doing another memory allocation and copying the value over, made the language even more powerful. The downside though, is that with more ability to handle memory meant more responsibility to handle it properly.

The Rule of Three was enhanced, and so if your class did any of the following:

destructor

copy constructor

copy assignment operator

move constructor

move assignment operator

The Rule of Five states that if you coded at least one of them, then you should code all 5 of them.

The next two terms are different but sometimes linked together. Resource Acquisition Is Initialization is a programming technique that says you must safely acquire the resource before it can be used. One way of looking at this is, if required, lock your resource from outside use if you are modifying it. One reason why it is mentioned here is because the new move semantics made this technique a lot safer. There is also a Rule of if you also include a default constructor.

The other term is coined The Rule of in a way combining the Rule of Five and the RAII technique. It can be looked at in different ways, but it says that a resource should be managed within its type. External resources (outside of that particular class) should not be able to directly modify the memory of that resource, and that way, within the resource's class, it can use RAII to safely manage how and when the memory is used. Within that

resource's class, it should still follow The Rule of Five, but the consumers of that class/resource do not have to write extra code to manage it and do not have to code those functions. If required, they can also set the functions to default.

Here are the header files we will use in this chapter:

```
#include

#include

#include

#include

#include
```

We have a code sample that shows the move operation, and our concept:

```
{




Restaurant() =

// opening a new restaurant

name)
```

```
{

<< "Creating restaurant with

ChangeName(name);

}

name) {

ClearName();

 = +

 _Name =

#ifdef _MSC_VER

strcpy_s(_Name, _NameSize, name);

#else

name, _NameSize);

 =

#endif
```

```
}

// creating another chain restaurant

Restaurant& other) {

<< "Creating restaurant via

  = other._NameSize;

  _Name =

#ifdef _MSC_VER

strcpy_s(_Name, _NameSize, other._Name);

#else

other._Name, _NameSize);

  =



#endif

}
```

```
// changing ownership via move ctor

Restaurant(Restaurant&& other) noexcept {

<< "Creating restaurant via

  = other._NameSize;

  _Name = other._Name;

other.ClearName();

}

// changing ownership via move assignment

Restaurant& other) noexcept {

if == &other) {

<< "assignment op, but for same entity,

  }

else {

<< "assignment op for restaurant via
```

```
  = other._NameSize;

  _Name = other._Name;

other.ClearName();

  }

return

}

~Restaurant() {

if != {

<< "Destroying restaurant w



  }

else {

<< "Destroying restaurant with no
```

```
  }
}

_Name;
uint32_t
= {
if {
_Name;
  }
  =
  _Name =
}
};
{
```

```
Restaurant& other) : _restaurant(other) {

}

// move ctor

RestaurantOwner(Restaurant&& other) :

// _restaurant(other)  // would still copy

// _restaurant((Restaurant&&)other) // works but messy cast

  {


}


Restaurant _restaurant;

};

{
```

{

of my

Restaurant place = "Hello

// Restaurant dest = place; // but will copy

// Restaurant dest = (Restaurant&&)place; // works with move ctr, but messy

// Restaurant dest((Restaurant&&)place); // works with move ctr, but messy also

Restaurant

Restaurant anotherPlace = "Hello

// Restaurant dest2 = std::move(place);

// even though using assignment operator, it knows it is the ctor and will call it

Restaurant dest2 =

dest2 =

```
}

};
```

The name does allocate memory, but that is done complementarily within the Restaurant class. Nonetheless, here we have a small example: If we transfer ownership of the restaurant, we are not making a copy of the whole place. Rather, the resources (in this case just the name) are being moved, or to be more precise, the ownership of the memory is being transferred, and the building stays in the same place. We did not make a copy of all the contents of the restaurant; we just moved the ownership over. The output can be seen in the following Figure

Creating restaurant with arg
Creating restaurant via move
Destroying restaurant with no name
Creating restaurant with arg
Creating restaurant via move
Creating restaurant with arg
Creating restaurant via move
assignment op, but for same entity, no-op
Destroying restaurant w delete
Destroying restaurant with no name
Destroying restaurant w delete
Destroying restaurant with no name
Destroying restaurant w delete

Figure 5.1: Output from LookAtObjects

We can see our move functions are working as we would hope. We transferred the name over, and when the temporary object gets destroyed,

it knows not to delete that resource because it was transferred over. Our code also does a little check to ensure we do not cause problems if we try to assign an object to ourselves.

It is to be noted that as the commented-out lines say, we could have performed the same operation by a cast, but aside from looking messy, it might not be obvious on the intent. Using a std::move method is cleaner, shows the intent, and will actually work in some places the cast would not work.

Yet another point to be noted is that you will see that we did an assert, and that our Restaurant class indeed does abide by the std::movable concept. It is a handy concept to determine or ensure that a class has that powerful feature.

Recipe 5.2: copyable

Problem: We want a concept to tell us if a type supports the copy constructor and copy assignment.

Solution: In addition to std::movable from C++ 20, there is also

The copyable concept needs the movable concept, in addition to copy_constructable and some assignable_from checks.

We did not have to have as much of a preamble for this concept, as it is mostly self-explanatory, or already covered in the last topic. We have a smaller code sample looking at the checks:

```
{



{



Coffee() =

name) {}

// if no copy-ctor, then failures Problem1, Problem2, Problem5
```

```
// RichCoffee(const RichCoffee& other) = delete;

Coffee& other) {}

//RichCoffee(const RichCoffee& other) = default;

// if no copy=, then failures Problem2, Problem3, Problem4, Problem5

//RichCoffee& operator=(RichCoffee& other) noexcept = delete;

//RichCoffee& operator=(RichCoffee& other) noexcept {}

  Coffee& other) noexcept =


// if no copy =const, then failures Problem2, Problem4, Problem5

//RichCoffee& operator=(const RichCoffee& other) noexcept = delete;

//RichCoffee& operator=(const RichCoffee& other) noexcept {}

  Coffee& Coffee& other) noexcept =

// if no move ctor, then failures Problem1, Problem2, Problem5

//RichCoffee(RichCoffee&& other) noexcept = delete;
```

```
//RichCoffee(RichCoffee&& other) noexcept {}

Coffee(Coffee&& other) noexcept =

// if no move =, then failures Problem1, Problem2, Problem3, Problem4, Problem5

//RichCoffee& operator=(RichCoffee&& other) noexcept = delete;

//RichCoffee& operator=(RichCoffee&& other) noexcept {}

  Coffee& other) noexcept =

};

{




Coffee&>,

const Coffee&>, "Problem4.assignable_from



}
```

```
};
```

The class does not have any implementation, but the point was to see what was required to satisfy the concept. It is to be noted that when building this, the compiler was not always being consistent about what assertions failed.

Sometimes, if we comment out the function, it would not fail, and so, we have to explicitly mark the function as deleted, to trigger the failures. Obviously, if we remove the move constructor, then movable and copyable should fail. Some experimentation can be done in your own code, but aside from showing the requirements of the copyable concept, the other point is to review your code and whether there are times when you want/need to add in a default operator, to ensure you meet the wants of a concept. Conversely, although quite rarely, there might be times when you want to explicitly mark a function as deleted, and then you would have to determine the repercussions of something not satisfying a concept.

Recipe 5.3: semiregular

Problem: We want a check for something that has the requirements of copyable, plus has a default constructor.

Solution: C++ 20 included a concept named semiregular.

Conversely, with the previous chapter, where we went through several topics without many code samples until the later stages, we started with a big example, and now some of our subsequent concepts just build on top of that, but there is not much new to show.

The semiregular concept uses the copyable that we just learned, as well as That equates to the same copy and move functions, plus a constructor that does not take any parameters.

We added to our previous code sample dealing with coffee:

```
{



: public Chapter5_2::Coffee {



EnrichedCoffee() =
```

```
  =

other) =

  EnrichedCoffee& other) noexcept =

  EnrichedCoffee& EnrichedCoffee& other) noexcept =

EnrichedCoffee(EnrichedCoffee&& other) noexcept =


  EnrichedCoffee& other) noexcept =

};

{

// compiler error, although we knew regular Coffee would not work

// static_assert(std::semiregular);


// compiler error, due to no default constructor for Coffee

// static_assert(std::semiregular);
```

```
}

};
```

Our EnrichedCoffee public derives from and we specified our operators. On some compilers, we might get an error. That is because we have a default EnrichedCoffee constructor, but the basic Coffee constructor is deleted, so the compiler may not know what to do. If it works, then good, but we should clean that up so it is clearer what it is doing.

If we go back in and tweak the code, we have:

```
{



: public Chapter5_2::Coffee {



EnrichedCoffee() : {}

  =

other) =

  EnrichedCoffee& other) noexcept =



  EnrichedCoffee& EnrichedCoffee& other) noexcept =
```

```
EnrichedCoffee(EnrichedCoffee&& other) noexcept =

  EnrichedCoffee& other) noexcept =

};

{

//

// compiler error, although we knew regular Coffee would not work

// static_assert(std::semiregular);



}

};
```

The compiler now sees that we have the right time of constructure, which now calls the base constructure with an argument. Now our semiregular constraints have been met.

Recipe 5.4: regular

Problem: We need a concept that does equality comparison, in addition to all specified in the semiregular.

Solution: We are not sure of the phrasing of why equality comparison turns something from semiregular to regular, but regular was also introduced in C++20.

Our concepts in this chapter certainly are getting bigger and more powerful. The concept regular also requires for the templated type to satisfy which we covered in a previous chapter. This only means one extra operator function, but we will start with a new class, just to be clear:

```
: public Coffee {



EnrichedCoffee() : {}

//EnrichedCoffee() = default;

~EnrichedCoffee() =

other) =

EnrichedCoffee& other) noexcept =
```

EnrichedCoffee& EnrichedCoffee& other) noexcept =

EnrichedCoffee(EnrichedCoffee&& other) noexcept =

EnrichedCoffee& other) noexcept =

};

: {

BavarianRoastedCoffee() =

~BavarianRoastedCoffee() =

BavarianRoastedCoffee& other) =

BavarianRoastedCoffee& other) noexcept =

BavarianRoastedCoffee& BavarianRoastedCoffee& other) noexcept =

BavarianRoastedCoffee(BavarianRoastedCoffee&& other) noexcept =

BavarianRoastedCoffee& other) noexcept =

```
BavarianRoastedCoffee& other) const { }

};

{

#pragma region normal coffee



Coffee&>,

const Coffee&>, "Problem4.assignable_from



#pragma endregion

#pragma region enhanced coffee

//

// compiler error, although we knew regular Coffee would not work

//static_assert(std::semiregular);
```

```
#pragma endregion enhanced coffee




}
```

This time we have no surprises. Our BavarianRoastedCoffee has that operator, and so it meets as well as our coffee being regular.

It is to be noted that when we are looking at a concept requiring something we just covered, we might incorrectly say that it derives from a concept. Concepts cannot derive from another like classes can; that would just be a common phrase. You might also read something like: the regular concept subsumes semiregular, which is the same thing we are talking about here. A new concept requires all of the constraints defined by something else, and most likely a bit more. If you recall in the first chapter, you could have || (or) clauses in your requirements. In that case, it is debated whether it would subsume something else, or if it is going to start stating other cases.

Conclusion

Although the number of concepts in this chapter was fewer, they are quite powerful and important in their rights. Having your classes follow proper resource-handling rules is critical; it is the difference between an application that leaks memory and/or crashes, versus one that runs well and is very efficient. Checking your types for these capabilities will elevate your code.

Points to remember

The differences, although sometimes small, occur between these object concepts.

Follow the “Rules” on how resources should be handled.

Utilizing these concepts will help your code to be sturdy and performant.

References

CPPreference. (n.d.). Retrieved from https://en.cppreference.com/w/cpp/language/raii

LinkedIn. (n.d.). Retrieved from https://www.linkedin.com/pulse/c-core-guidelines-rule-zero-five-six-rainer-grimm

YouTube (n.d.) Retrieved from https://www.youtube.com/watch?v=OWNeCTd7yQE

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 6

Using the New Callable Concepts

Introduction

We have covered a variety of groups of concepts, and callable concepts is the last of this main group of chapters on concepts, although we will certainly be talking more about them later.

In a broad sense, a callable is something that one can call to execute some functionality. For our C++ purposes, it would be a type for which one could invoke. There are different entities one can invoke, and there are different mechanisms in how you can invoke them. We will go into some detail, (but not too deep), so that we can talk about how to set up some of the concepts. We have several concepts that we are going to talk about, but some of them are not directly used that much, and some are more related to ranges. Just like anything else related to concepts, it will be good to at least learn about them, in case you need to set something up with them. Certainly, C++ is very different today, versus how it looked in the pre-modern era. There is a lot more focus on organization, and defining things in a way that they can be flexible. The goal is to keep classes small, functions small, and utilize the power the language, the compiler, and the operating system offers us. That means being able to take a small piece of code, and to define that we have choices on how to execute that code. It could be to execute it right away, or we could execute it later, after something else gets completed. That something else could be unrelated, by tasks that are queued up so we do not over-burden the system, or that something else could be related, and we are saying that to execute some code, something else has to happen which puts us in the right state. All of these choices means there is more to understand, but also gives us great power on how we want our program to run.

Structure

In this chapter, we will cover the following recipes:

Recipe 6.1: invocable

Recipe 6.2: regular_invocable

Recipe 6.3: predicate

Recipe 6.4: relation, and equivalence_relation

Recipe 6.5: strict_weak_order

Recipe 6.6: indirectly_unary_invocable, and indirectly_regular_unary_invocable

Recipe 6.7: indirect_binary_predicate

Recipe 6.8: indirect_equivalence_relation

Recipe 6.9: indirect_strict_weak_order

Admittedly, this chapter has more theory and examples, but it would be a good thing to go through all of these concepts. It seems the major

compilers have these features covered.

}

Objectives

This chapter will review what callables are. We will also talk about the various new concepts that relate to the callables, and how best to use them. They are something that we often use, but sometimes do not appreciate the benefits they offer. They provide flexibility to your code, and a good means of defining a unit of work.

Recipe 6.1: invocable

Problem: We need an easy way of checking a callable, that can be called with a specific set of argument types.

Solution: C++ 20 has a new concept named

When we say that the concept invocable ensures a callable can be called correctly, you might easily understand that statement, or you might ask a couple of questions, such as:

What is a callable?

What does it mean to call it correctly?

We will recap a bit before getting deep into our concepts. If we want to execute some code, there are a number of ways to do that, and some of them are:

Using a lambda; some anonymous code we have defined.

A pointer to a member function.

A free function, otherwise called a non-member function.

When you overload the operator() on your type, use a functor so that you can execute some code by saying you are calling the object itself. This is

however, not normally recommended.

One might argue to include but that is more of a type-erased wrapper around a callable, versus a callable itself. You can declare a std::function with a lambda, but it will not do anything by itself.

For our code samples in this chapter, we will use the following headers:

```
#include

#include

#include

#include

#include
```

If we create a little code sample, this looks at some calorie counts for some meals:

```
concept Adjuster =
```

```
{


calories, Adjuster auto adjuster) {

for cal : calories) {

<< <<

  }

}


{

<< << << __FUNCTION__ <<

calorieCounts{ 4000 };


PrintModifiedCalorieCounts(calorieCounts, v) { return v + });

}

};
```

You can see we have defined our concept which is an invocable that takes an int parameter. We created a little collection of some recent calorie counts, and we call the function The parameters for this are a collection and some function that will be executed while it loops through the collection. In our case here, it is a simple lambda that increases the value by 10. Looking at the output in Figure that seems fairly straightforward:

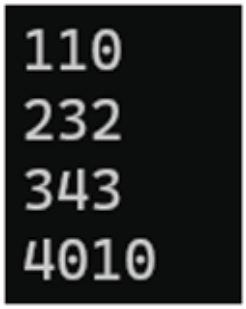

Figure 6.1: Output from LookAtCalorieCounts

If we go back and look at our Adjuster concept, it says that it takes some executable code and has a parameter of an integer. The invocable concept can take many parameters, but we kept it to the one for simplicity. After we call the adjuster code, we print out the result. The curious question would be: how did it know the return type? Although possible, since it was just outputting the value to it did not care. The other question though, is how did the invocable know what the output type was going to be?

We will slightly modify our print function, and call it a few more times:

```
// reused from previous section


concept Adjuster =
```

```
value) { return value % }

{

value) { return value * }

};

{


calories, Adjuster auto adjuster) {

for cal : calories) {

<< << ",

  }

<< <<

}


{

<< << << __FUNCTION__ <<
```

```
calorieCounts{ 4000 };


PrintModifiedCalorieCounts(calorieCounts, v) { return v + });

auto adjuster1 = v) { return v * };

PrintModifiedCalorieCounts(calorieCounts, adjuster1);

auto adjuster2 = v) { return v / };

PrintModifiedCalorieCounts(calorieCounts, adjuster2);

auto adjuster3 = v) { is:" + / + };

PrintModifiedCalorieCounts(calorieCounts, adjuster3);

PrintModifiedCalorieCounts(calorieCounts, CheckIfOdd);

PrintModifiedCalorieCounts(calorieCounts, SomeAdjuster{});

}

};
```

We only modified PrintModifiedCalorieCounts to print the values on the same line, just to save space. We created a few more lambdas to call and assigned them to variables. We also are calling a static (non-member)

function, as well as a functor. The functor is SomeAdjuster's operator() overload and we created an instance of that struct to pass into the print function. The output is shown in Figure

```
110, 232, 343, 4010,

200, 444, 666, 8000,

23.8095, 52.8571, 79.2857, 952.381,

# is:50.000000!, # is:111.000000!, # is:166.500000!, # is:2000.000000!,

0, 0, 1, 0,

200, 444, 666, 8000,
```

Figure 6.2: Output from LoookAtCalorieCounts

The output values are not overwhelming; the output seems to be what we expected based on the input. We notice that the invocable only cares that we pass in an integer, it was not caring what the output type was. In our code here, it accepted the double, the string, and the Boolean without complaint. If you recall in our earlier chapter when talking about concepts, some are more focused on the existence of a member variable, and do not care about the type. For this is also the case. It just wants to ensure it call execute some code, and it is not worried about what the output will be.

Suggestion

As an aside, we will just spend a minute talking about calling a callable. In most cases in this chapter, for simplicity's sake, we will just use lambda. We could use std::invoke as well, but passing the lambda was easier. There are some cases where std::invoke is cleaner and easier to observe. It is also easier to use std::invoke for pointers to class member functions. Plus, with std::function you can most likely avoid the need to do std::bind as well. You can still use std::bind if you want, but std::function works with lambdas, member functions, and other mechanisms, so they are more flexible.

Recipe 6.2: regular_invocable

Problem: We want something similar to invocable but also equality preserving.

Solution: In C++ 20, there is also the concept

Our next concept, unsurprisingly builds off of The only extra requirement is that regular_invocable requires equality preservation.

Let us look at what that means. Sometimes, it is not easy to explain, or to understand. To preserve equality on an expression means that you should get equal output when provided equal inputs. The inputs to an expression are the operands you provide. The outputs of an expression are the result(s) of whatever operands that were modified. If an output is affected by recursion, or multi-threading execution, or some other time-based value, then you probably cannot guarantee you will get the same output.

If your concept abides by this, then you could use regular_invocable as your requirement instead. If we quickly change our concept:

```
template <

concept Adjuster6_2 =

{
```

```
calories, Adjuster6_2 auto adjuster) {

for cal : calories) {

<< << ",

  }

<< <<



}



{

<< << << __FUNCTION__ <<

calorieCounts{ 4000 };

PrintModifiedCalorieCounts(calorieCounts, v) { return v + });

auto adjuster1 = v) { return v * };

PrintModifiedCalorieCounts(calorieCounts, adjuster1);

auto adjuster2 = v) { return v / };
```

```
PrintModifiedCalorieCounts(calorieCounts, adjuster2);

auto adjuster3 = v) { is:" + / + };

PrintModifiedCalorieCounts(calorieCounts, adjuster3);

PrintModifiedCalorieCounts(calorieCounts, CheckIfOdd);

PrintModifiedCalorieCounts(calorieCounts, SomeAdjuster{});

}

};
```

Everything still compiles properly, and we get the same output as shown in Figure

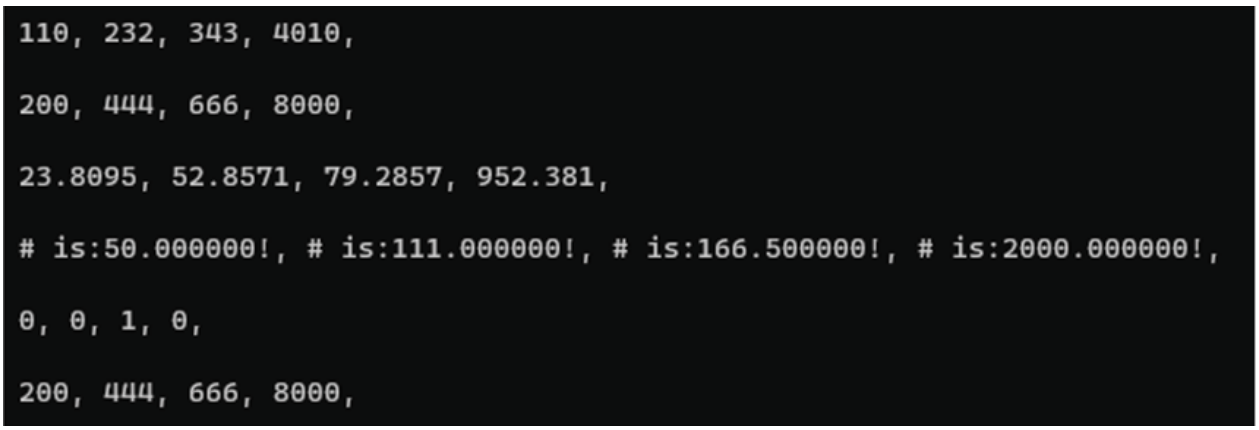

Figure 6.3: Output from LoookAtCalorieCounts, after modifying concept

Recipe 6.3: predicate

Problem: There are many times when we need a way of checking whether a callable returns true or not.

Solution: The next concept we will look at, is called

The concept predicate continues down a familiar path, building off of previous concepts. This time, predicate is the same as but ensures the callable returns a Boolean. Up until now, we have not been concerned about a return type, but it makes sense in requiring a Boolean result. If you recall from our earlier chapter when introducing concepts, the requires clause lists predicates and only wanting to know if they are true or false.

We can show this in a code sample similar to before:

```
=

{


calories, LowChecker auto adjuster) {

for cal : calories) {
```

```
if {

<< cal << ",

  }

  }

<< <<

}


{


<< << << __FUNCTION__ <<

calorieCounts{ 333 };

= v) -> bool { return v < };

PrintModifiedCalorieCounts(calorieCounts, checkIfLow);

}
```

```
};
```

This time, our concept LowChecker uses a Our print loop calls the predicate and only if it returns true, do we print the result. We have our vector of integers, and we define our predicate checkIfLow returns successfully if it is below an arbitrary number of 70. Of course, this same result can be done in many ways, but it is good to know we can involve a concept in a solution.

Recipe 6.4: relation and equivalence_relation

Problem: We want a way of checking the relationship between multiple expressions.

Solution: C++ 20 was released with concepts named relation and

We will only mention these concepts; they are used more as building blocks for other concepts. They specify that a relation defines a binary relation over a set of expressions. If you have a set, you can say two objects in that set are related if they satisfy a certain property.

The concept equivalence_relation is similar to but it does mention some semantic requirements:

A relation r is an equivalence relation if

It is reflexive: for all x, r(x, x) is true;

It is symmetric: for all a and b, r(a, b) is true if and only if r(b, a) is true;

It is transitive: r(a, b) && r(b, c) implies r(a, c).

The relation concept ensures that you can apply an algorithm that can tell if there is a relation between the objects, while the equivalence_relation is more precise, and states that the two objects are equivalent.

Recipe 6.5: strict_weak_order

Problem: We want a way to impose a strict weak ordering on its arguments.

Solution: Related to relation, there is also the strict_weak_order concept.

Again, a concept related to the relation concept, but semantic differences (See references):

A relation r is a strict weak ordering

It is irreflexive: for all x) is false;

It is transitive: for all b and if r(a, b) and r(b, c) are both true then r(a, c) is true;

b) be !r(a, b) && !r(b, then e is transitive: e(a, implies

Since these deal with math terms that could get into more depth than we want to cover in this book, let us borrow an analogy (See references) and relate it to our cooking theme. Say there was a pie-baking competition, where the baker had to cook a pie from scratch and present it on a plate to the judges before the timer expired. Of the 10 contestants, 3 of the bakers failed to present their pie on time, and so, were disqualified. There was one pie that the judges unanimously agreed was the best pie they tasted in

years. Then two other bakers made very good pies and the judges had a tough time deciding.

For weak ordering, it is easier to group them:

The pie that was unanimously the best.

The two pies were also very good.

The rest of the pies from the bakers were not disqualified.

The entries from the bakers were disqualified.

Thus, we do not know the complete ranking, but this weak ordering gives us a decent idea on how the bakers performed.

If all the bakers finished, and the judges could rank all the pies well enough that there were no ties, then that would be more of a strict weak ordering.

Another way of looking at it (See references), is that partial ordering would use the <= comparison operator to compare the values in the set. You could not perform a total order operation (such as because it would not know how to ensure things were ranked properly. Strict weak ordering is more like using the < operator when trying to sort your items. This would allow for greater precision. Let us say that you came across a comparison, for example, a situation where object a versus object b. When comparing if a < b is false, and b < a is false, then the only logical

conclusion you could make is that a and b are equivalent. There is a chance a and b and not identical, but in terms of sorting the set, that is the best conclusion you can reach. That would be called strict weak ordering.

Moreover, to clarify something, the strict part of the name means it is the irreflexive form < of the comparison, rather than the reflexive The weak part means that if a < b and b < it does not imply they are equal but comparatively equivalent.

Now that we have reached the end of our math lesson, we can say that the concept strict_weak_order complies with the requirements that an algorithm could be applied to a set so that it is sorted to the best of the algorithm’s abilities.

Recipe 6.6: indirectly_unary_invocable, and indirectly_regular_unary_invocable

Problem: We want a way of specifying requirements that call unary invocables.

Solution: We have the concepts and

These concepts combine some concepts that we have talked about before. They specify requirements that call unary invocables. A unary operator is one of the following symbols or the sizeof() function:

- (minus, or negative)

++ (increment)

-- (decrement)

! (logical not)

& (address of operator)

sizeof() function

As we learned earlier in this chapter, the difference between invocable and regular_invocable is the latter has equality preservation, which is the same

for the case of

These concepts are included in the header and not since they deal with iterators and are used with ranges, which we will delve into in a later chapter.

Recipe 6.7: indirect_binary_predicate

Problem: We want to specify requirements for an algorithm that calls binary predicates as their arguments.

Solution: There is

This is another concept that is also defined in the header, which we will just mention briefly. The concept indirect_binary_predicate is very similar to which we talked about earlier in this chapter. The main difference is that since it is indirect, it is applied to the references of the arguments, versus the arguments themselves.

Recipe 6.8: indirect_equivalence_relation

Problem: We want a way of checking the relationship between multiple expressions indirectly.

Solution: The concept indirect_equivalence_relation was released in C++ 20

There is a pattern happening with the concepts in this chapter. The concept indirect_equivalence_relation is similar to equivalence_relation (which of course, was similar to Again, the main difference here is that since it deals with the value indirectly, the algorithm is applied to the references of the arguments, versus the arguments themselves.

Recipe 6.9: indirect_strict_weak_order

Problem: We want a way to impose strict weak ordering on what the arguments are referencing.

Solution: C++ 20 has a concept named

Finishing off our chapter of callable concepts, although this too is defined in the header, the concept indirect_strict_weak_order is similar to except that the callable applies to what the arguments are referencing.

Conclusion

We have covered a fair bit in this chapter. We dove down to talk about various aspects of a callable (or invocable, if you prefer). We went off on a couple of tangents into sorting, but it is good to understand what the meaning is behind some of these concepts. They offer checks to ensure the code is done properly, and it should also help your code be cleaner as well, which is not always easy to do with callables. We are going to move away from concepts for now, to talk about const-related specifiers.

Points to remember

Three are different types of callables, some are easier and/or better to use than others.

There are different ways to call a callable, although there are both advantages and disadvantages to them.

There are many callable concepts; they were created to make your code a bit cleaner, but mainly as a check to ensure things are being done properly.

Some concepts care about the return type, some do not.

There are many related concepts: some that deal with the values directly, and some are for when the value is used indirectly through a reference.

References

CPPStories. (n.d.).

Quora. (n.d.).

CPPReference. (n.d.).

CPPReference. (n.d.).

Wikipedia. (n.d.).

StackOverflow. (n.d.).

StackExchange. (n.d.).

YouTube. (n.d.).

Join our book’s Discord space

Join the book’s Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 7

Const Related Specifiers

Introduction

The C++ keyword const has existed since C++ 98, and it helps the coder know when a variable or parameter can be modified. The C++ world of const-ness did not change much, until C++11 with the keyword It had improvements in the C++ 14 release, and again how the language dealt with const-ness did not change for several years. In C++ 20, not only were there further improvements to but two other keywords, consteval and were introduced. These added great power to the programmer, but more so for the compiler. They help the compiler understand what it can optimize and compile down to make the code run faster. What went from a simple mechanism, to denote a value could not be changed, to several mechanisms that can work together to not only ensure values are not changed but significantly speed up the performance of the code.

Structure

In this chapter, we will cover the following recipes:

Recipe 7.1: constexpr enhancements

Recipe 7.2: constexpr for transient allocation

Recipe 7.3: consteval

Recipe 7.4: constinit

Recipe 7.5: constexpr if

Recipe 7.6: if consteval

Recipe 7.7: Some hidden problems with constexpr

Unfortunately, the features being discussed are not fully implemented by all of the three major compilers. You might have to do workarounds, or wait to use some of them:

{

```
Chapter7_1_1::PrintThePotatoCount();//



Chapter7_2::PrintThePotatoCount();









}
```

Objectives

In this chapter, we hope you get an understanding of the new const-related keywords that are in C++. We will learn what they are, how to use them, and when or how we should not use them.

Recipe 7.1: Looking at the constexpr enhancements

Problem: We want to evaluate a variable, or the result of a function, at compile time.

Solution: There have been enhancements to constexpr in subsequent releases, including C++ 20.

While the keyword const was more for code usage, the keyword constexpr (which stands for constant expression) included in C++11, was more for code performance. The logic was (and still is) that if you can move some functionality from runtime to compile time, then that will have your compiler take a bit longer, although the code that is executed will run faster.

Before getting too much into our topic, we will acknowledge a couple of great helpful resources:

Compiler Explorer is an online compiler. Not only can you work in a sandbox, but you can also select different compilers and set the compiler setting. It also shows the assembler output of your code, so that you can get an idea of what your code is compiled into. Some of the figures provided in this chapter will be output from this tool.

C++ Weekly is a YouTube channel that presents lots of insightful content. The episodes often use Compiler Explorer, and they spend a bit of time looking at the assembly output, and how the compiler setting affects the output. They do

not get too deep into the assembly code, but enough for one to understand and appreciate what is going on.

Let us start by including the header files we will be using:

```
#include

#include

#include

#include

#include

#include

// for now we are ignoring the warning

#pragma warning( disable : 4244 )

#ifdef __clang__

COMPILER =  __VERSION__;

#elif __GNUC__

COMPILER = "gcc: " __VERSION__;
```

```
#elif (_MSC_VER)

COMPILER = '' +

#endif
```

We also included an extra variable to see what compiler you are using; in case you are testing it out with the Gnu and Clang C++ compilers. We also disabled a warning, not that we will recommend doing that a lot, but for now, we wanted to ignore some of the casting/conversion warnings so we could focus on the const-related issues.

If we create some sample code for a simple situation: we are having some guests over for dinner, and one of the dishes is mashed potatoes. A rule of thumb was to use one potato for every person, and one for the We are using a bit more math, suggesting cooking an extra potato for every five people We created a simple function and did some calculations:

```
{



people) {

return people + (people /

    }

{
```

```
guests =

household =

unexpected =

auto probablyPotatoCount =
HowManyPotatoesRequiredForMashed(household + guests);

auto maybePotatoCount = HowManyPotatoesRequiredForMashed(household
+ guests + unexpected);

Chapter7_1_Average =

auto potatoCount = (probablyPotatoCount + maybePotatoCount) /
Chapter7_1_Average;

auto safePotatoCount = + maybePotatoCount) / Chapter7_1_Average);

<< "probable=" << probablyPotatoCount << ", maybe=" <<
maybePotatoCount << ", result=" << safePotatoCount <<




    }



{
```

```
<< << << __FUNCTION__ << ", for " << COMPILER <<

auto number = LookAtPotatoCount();

<< "The potato count is: " << number <<

    }

};

{




}
```

The other thing to consider is that we are not sure of the exact number of people. There are four in the family, plus five confirmed guests, and maybe two others. So, our code takes the confirmed attendee result, and the maximum attendee result, and then takes the average, rounded up, as shown in Figure

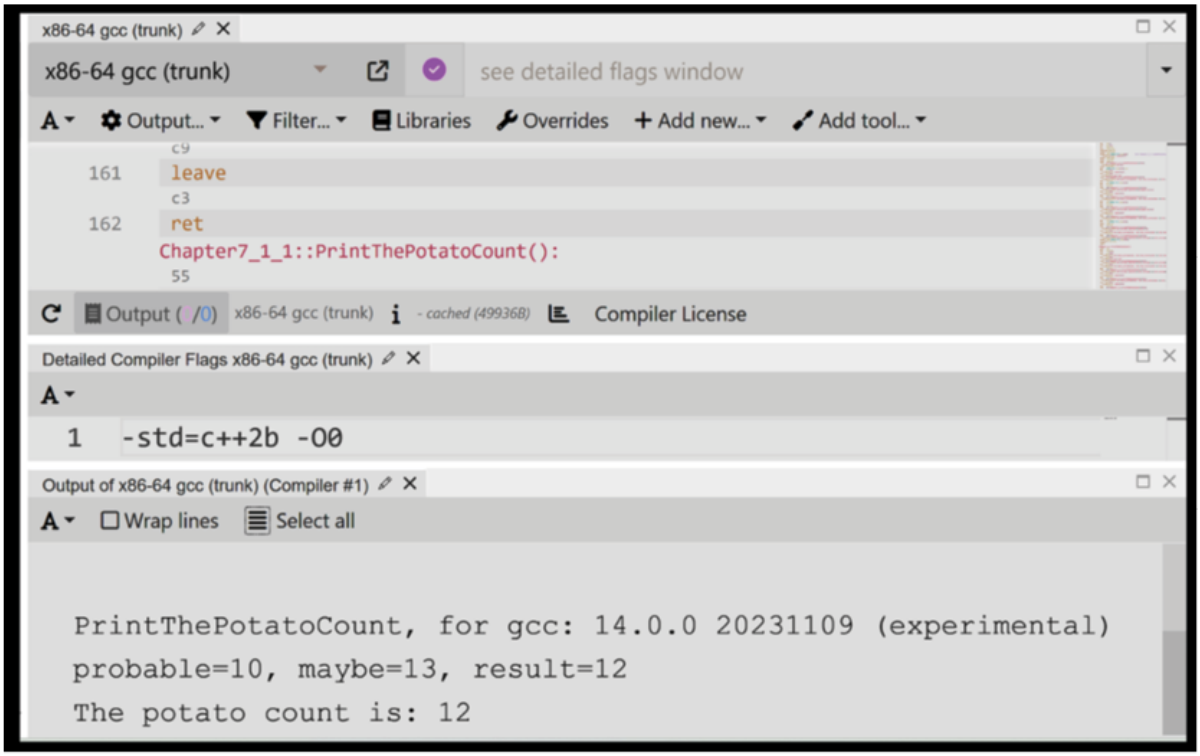

Figure 7.1: Recipe 1 Godbolt output for no optimization

A few things to note in the preceding figure:

In the top pane, we can see it is currently set to which means there is no optimization.

We can see the bottom of the output panel, our output says that to be safe, we should use 12 potatoes.

In the middle of the image, it shows how many assembly instructions there are. We certainly do not need to get technical, whether they are actual assembly instructions; we are just looking for a rough count. The count is in hex, and you can see we have quite a lot of instructions for calling a small method. We are starting with a program that has to execute 354 instructions. The numbers can vary between environments, and even per build of the

compiler, so your results may vary. We are just looking at the number as an estimate to track our progress.

Now that we know our code works, to make things cleaner, we will just comment out our cout lines:

```
{



people) {

return people + (people /

    }

{

guests =

household =

unexpected =

auto probablyPotatoCount =
HowManyPotatoesRequiredForMashed(household + guests);

auto maybePotatoCount = HowManyPotatoesRequiredForMashed(household
+ guests + unexpected);
```

```
Chapter7_1_Average =

auto potatoCount = (probablyPotatoCount + maybePotatoCount) /
Chapter7_1_Average;

auto safePotatoCount = + maybePotatoCount) / Chapter7_1_Average);

//std::cout << "probable=" << probablyPotatoCount << ", maybe=" <<
maybePotatoCount << ", result=" << safePotatoCount << std::endl;


    }



{

//std::cout << std::endl << std::endl << __FUNCTION__ << ", for " <<
COMPILER << std::endl;


//std::cout << "The potato count is: " << number << std::endl;

    }

};
```

That will get rid of string usage, and using the output stream, as shown in Figure

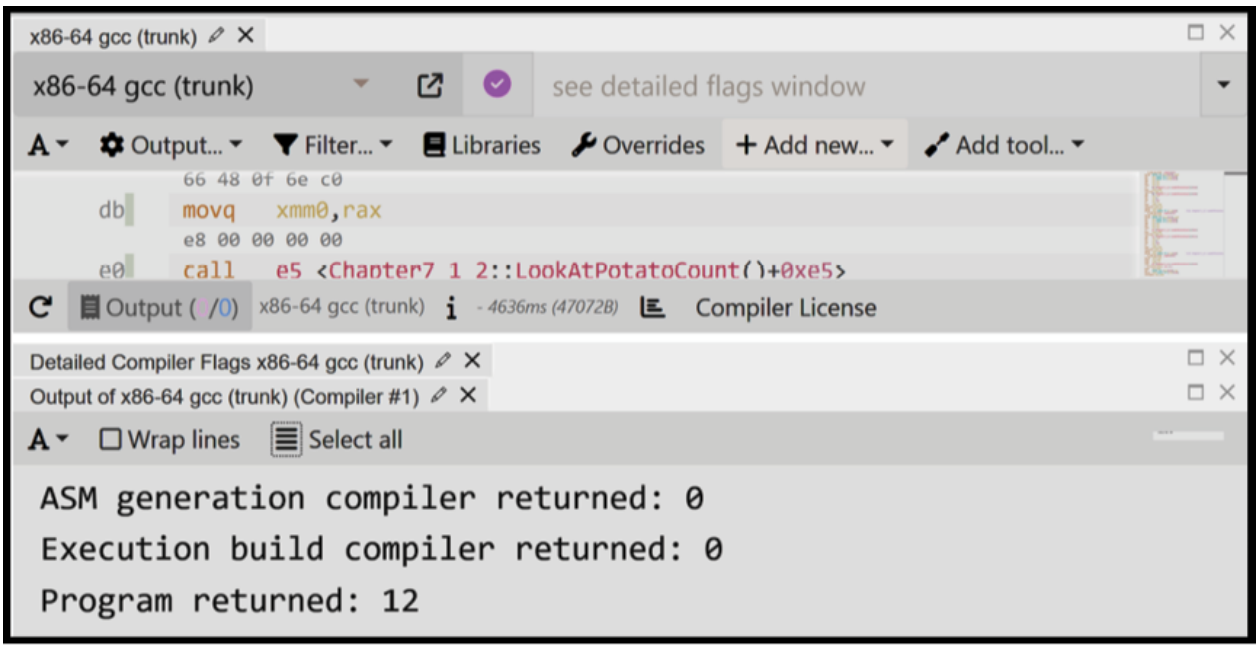

Figure 7.2: Recipe 1 Godbolt output for revised code

Understandably people will still need output; we are just trying to make our point easier to follow. Getting rid of the output shows that we have a lot fewer instructions to run, so now we have around 224 instructions.

Now we can get back to actually looking at our code. We hard-coded some inputs and called a simple function. Since we are not changing the input values, we can change them to be constant:

```
{
```

```
people) {
```

```
return people + (people /

    }

{

guests =

household =

unexpected =

auto probablyPotatoCount =
HowManyPotatoesRequiredForMashed(household + guests);

auto maybePotatoCount = HowManyPotatoesRequiredForMashed(household
+ guests + unexpected);

Chapter7_1_Average =

auto potatoCount = + maybePotatoCount) / Chapter7_1_Average);


    }


{
```

```
// std::cout << std::endl << std::endl << __FUNCTION__ << ", for " <<
COMPILER << std::endl;



//std::cout << "The potato count is: " << number << std::endl;

    }



};
```

Now after making those quick code changes, it reduced the number of instructions a little bit more. It is not worth showing the output for each step, but we now have 165 instructions. Our program is becoming pretty efficient, just by marking some things as constant. Even if it was not a huge gain, it would be a wise change to do anyway. The next thing to think about, is whether we can make our code run faster.

One of the goals with constexpr when it was released, was to make your code faster by moving work from runtime to compile time (when possible). Another goal would be to catch errors sooner, which is a great thing in itself. Obviously, in most cases, most work has to be done at runtime, but every little bit helps and if we can move some code to compile time, then it means when our program starts up, some calculations are already done. Our function is simple and seems like a good candidate, so we can make that code change.

```
{



people) {
```

```
return people + (people /
    }
num)
    {
#ifdef __GNUC__

#else

return == num)
            ?
            : + ((num > :
#endif
    }
{
guests =
```

```
household =

unexpected =

probablyPotatoCount = HowManyPotatoesRequiredForMashed(household +
guests);

maybePotatoCount = HowManyPotatoesRequiredForMashed(household +
guests + unexpected);

Chapter7_1_Average =

#ifdef __clang__

potatoCount = (probablyPotatoCount + maybePotatoCount) /
Chapter7_1_Average;

#else

potatoCount = + maybePotatoCount) / Chapter7_1_Average);

#endif

//std::cout << "probable=" << probablyPotatoCount << ", maybe=" <<
maybePotatoCount << ", result=" << potatoCount << std::endl;
```

```
    }


{

// std::cout << std::endl << std::endl << __FUNCTION__ << ", for " <<
COMPILER << std::endl;



// std::cout << "The potato count is: " << number << std::endl;

    }

};
```

Now, when we look at the number of instructions, we are down to 59. This is a sizeable improvement to reduce a lot of the instructions with just a few keywords. There are some situations where we should not equate fewer instructions with faster performance. In this scenario though, we are pushing some of the work to the compiler versus doing more at runtime. We can make the assumption here that fewer instructions would probably mean faster execution.

We have to mention a couple of other points, that not all compilers have completed their work on the latest features related to constant values. Only the gcc compiler seems to have updated its library to allow math functions (such as ceil to round off the value) to be There might be places as well, we you will have to do a workaround if a feature is not available. Even the clang compiler had some other issues, so albeit messy, we wanted to show what worked for us, and what did not.

A major factor, is the fact that most people never ship their product with no optimization. Compilers are very smart nowadays, and we let them do their best to make the program faster. You ask Is our code any better if we turn on the optimization for our code?

Refer to Figure

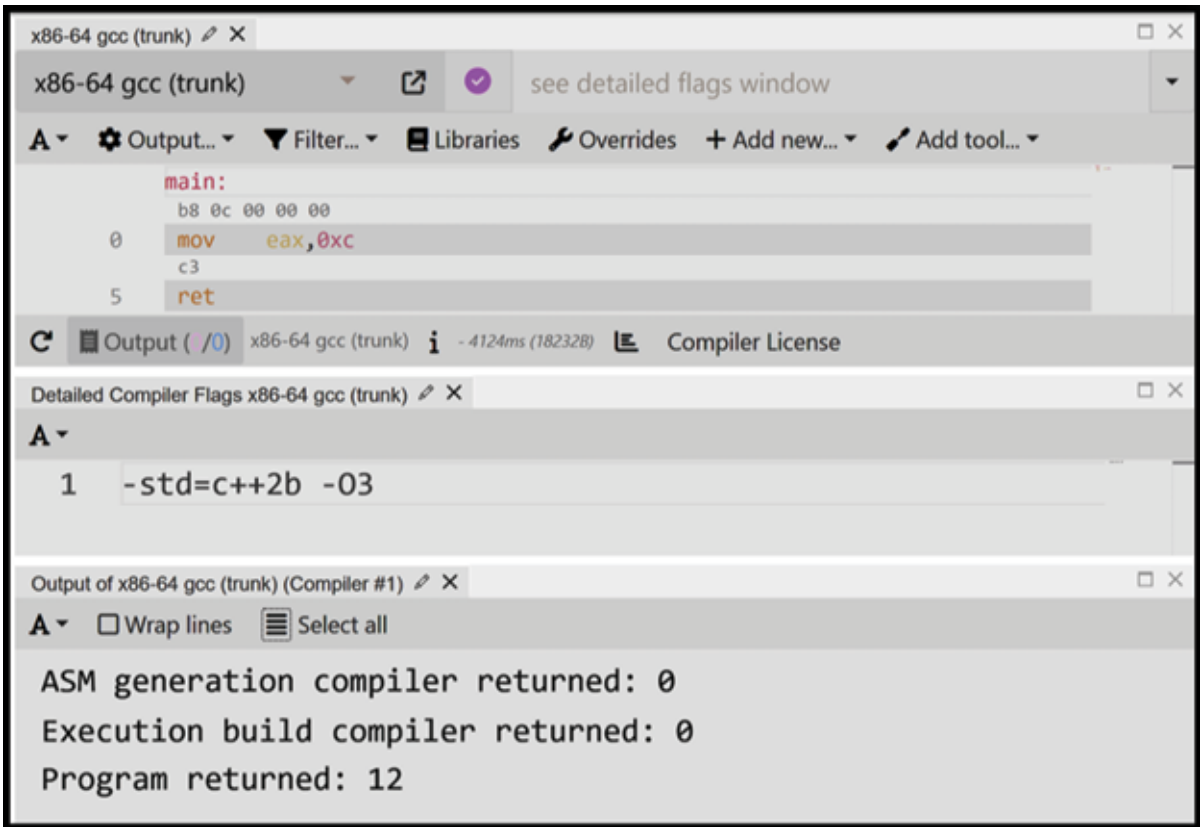

Figure 7.3: Recipe 1 Godbolt result after optimization

That assembly output window shows the whole application! With optimization, the compiler knew that the constant inputs are given to a method that could be run at compile time, and so the result is that all the program had to do was to return a pre-calculated result.

It means that the compiler will take a bit more time to find the best way to optimize, but normally your computer (or build server) is built to handle lots of work. The main thing is to make the end user happy. In this small example, we have seen a dramatic decrease in what the program will have to do, which means that the application will be a lot faster, and that will make the end user very happy. The goal of these const-related keywords is to move as much code as we can, so that the compiler can do more work, with less effort at runtime.

There is a lot more to talk about but another thing to talk about on this topic, actually relates to the topic title, and that is, what the constexpr enhancements are. The goal stayed the same since the feature was first released, but if we go back and look at our code, we want to see all that it did so it could actually compile down to a simple value.

The big enhancement comes from a lot of effort, looking at many standard library functions to if they too, could include the constexpr keyword so they could be run at compile time. Sadly, when constexpr was first released, the idea was great, but the end result was lacking, because in order for somebody's function to be run at compile time, meant that whatever library code that function called also had to be runnable at compile time. A lot of work has been done in recent years, especially for the C++23 release, to allow as many std functions as possible to be run at compile time, finally achieving the goal of more work done at compile time and less at run time. The list is too long to show, but when you look through the documentation, you will see many functions marked as constexpr since C++23 or constexpr since C++20.

Recipe 7.2: constexpr for transient allocation

Problem: We want to be able to temporarily allocate some memory in our constexpr functions.

Solution: Compilers have improved support for such a requirement.

As mentioned in the previous topic, although the idea of constrexpr was good, there have been many hurdles to overcome, to get to the point where the compiler could execute the code. A huge addition to C++20 was when the types string and vector were modified, so they could be executed at compile time.

Another hurdle to get to achieve goal of compile-time execution, was heap objects. Originally, people did a lot of workarounds to keep things on the stack. The thought was, that if it stayed on the stack, there was more of a chance of the code being optimized and able to execute at compile time. There are still some cases where that might be true, but thankfully things have greatly improved in recent releases.

If we revise our code a bit, as follows:

```
{



num)
```

```
    {
#ifdef __GNUC__


#else
return == num)
            ?


            : + ((num > :
#endif
    }
#ifndef __clang__
size_t count) {
weightedCount =
```

how can we check if our percentages add up to 100?

```
        0 };

for i = i < count; {

totalPercent +=

*

        }

long total = weightedCount->end(),



if {

<< "Math is a little bit off, should be 100%, but instead is: " <<
totalPercent <<

        }

/ (totalPercent *

    }

{

= { };
```

```
auto result =

return result;

    }

#endif



{

<< << << __FUNCTION__ << ", for " << COMPILER <<

#ifdef __clang__

<< "not implemented yet" <<

#else

auto number = UpdatedLookAtPotatoCount();

<< "The potato count is: " << number <<

#endif
```

```
    }

};
```

Running this with our Microsoft compiler, we get the output as seen in Figure

Figure 7.4: The output of PrintThePotatoCount

We are not recommending you write software to calculate the number of potatoes you need. Rather, we are doing something a bit different. We created a list of pairs; the number of people and the chance (as a percentage of 100) that this particular number of people will show up. In our function we are temporarily creating a container that will hold a We will then take the sum of those weighted counts to get the average. This actually shows two improvements that you could not do previously with

We intended to use a but some compilers still have some issues with that, so we are using a

Allocate memory in a constexpr expression.

If you notice, we properly deallocate (via the the transient allocation. Although allocating memory within a constexpr is great progress, some

compilers are still working on the implementation. Of course, these days you should be using std::make_unique and similar calls, but we just wanted to show you the progress when allocating in the older way. If you accidentally forget that delete line, some compilers are great and will generate an error telling you that you have forgotten to free the memory. That is great news, not only do you avoid a runtime leak, but the compiler itself is telling you, with extra tooling! The downside is that not all compilers have implemented that, and some do not have any errors or warnings yet.

An important point here is that you cannot return any allocated memory outside the Although painful at times, it does make sense, and you will need to ensure you follow that rule.

A side topic about this code; we provide a set of percentages to the function, and it should always add up to 100%. The question we have (with the TODO comment in the code), is what should we do in that case? The goal is that this code is executed at compile time, so we certainly cannot pop up a UI window telling the user about a data error. In some designs, some might have thrown an exception and it would have bubbled up to where we could alert the user. Now that the focus is compile-time execution, it is not ideal to throw an exception, since that will just fail the build. The decision on how to handle that situation is up to you, but something to be mindful of.

Recipe 7.3: consteval

Problem: We want a way to specify that a function must produce a compile time constant expression.

Solution: C++ 20 has the new keyword

A couple of important points that we have not talked about constexpr functions yet, is that although they must have the ability to be executed at compile time, they do not say they will be run at compile time. There is no mandate that the compiler must run the constexpr code, only that it could if it wanted or needed to. If functions are marked for they can be run at compile time or at run time. If we look at our code from an earlier example, if we knew the expected result and wanted to verify that, we might have a problem.

```
{



people) {

return people + (people /

    }

num)
```

```
    {
#ifdef __GNUC__


#else
return == num)
            ?


            : + ((num > :
#endif
    }
{
guests =
household =
unexpected =
```

```
auto probablyPotatoCount =
HowManyPotatoesRequiredForMashed(household + guests);

auto maybePotatoCount =
HowManyPotatoesRequiredForMashed(household + guests +
unexpected);

Chapter7_3_Average =

auto potatoCount = + maybePotatoCount) / Chapter7_3_Average);

// cannot do output at compile time

//std::cout << "probable=" << probablyPotatoCount << ", maybe=" <<
maybePotatoCount << ", result=" << potatoCount << std::endl;

// we would get a compiler error

// static_assert(potatoCount == 12);


    }


{

<< << << __FUNCTION__ << ", for " << COMPILER <<
```

```
auto number = LookAtPotatoCount();



<< "The potato count is: " << number <<

    }

};
```

Normally, one would verify using which demands compile time checking. In our case, it would fail, because the variables we used were just type auto. We did not say that had to be executed at compile time, but we were fortunate that the compiler did execute it. We will give the disclaimer that results might vary depending on your compiler and version (at the time of writing, only the gcc compiler worked), but if we update our code:

```
{



#ifdef _MSC_VER

people) {

return people + (people /

    }

#else
```

```
people) {

return people + (people /

    }

#endif

num)

    {

#ifdef __GNUC__



#else

return == num)

            ?

            : + ((num > :

#endif
```

```
}

{

guests =

household =

unexpected =

probablyPotatoCount =
HowManyPotatoesRequiredForMashed(household + guests);

maybePotatoCount = HowManyPotatoesRequiredForMashed(household +
guests + unexpected);

Chapter7_3_Average =

#ifdef __clang__

potatoCount = (probablyPotatoCount + maybePotatoCount) /
Chapter7_3_Average;

#else

potatoCount = + maybePotatoCount) / Chapter7_3_Average);

==
```

```
#endif

    }

{
<< << << __FUNCTION__ << ", for " << COMPILER <<
#ifndef __GNUC__
<< "only partially implemented" <<
#endif
auto number = LookAtPotatoCount();
<< "The potato count is: " << number <<
}
};
```

We did a couple of things; we marked our calculation method as saying this must be executed at compile time. Marking a function at consteval denotes that it is an immediate function, meaning it must be evaluated at compile time. We also added constexpr to our variables. After we did that, we can compile with our Variables and functions can be marked as since declaring a variable with a value is still an expression. Only functions can be marked as

At this point, we can add in a few different notes:

We have one function as static consteval and the other as constexpr There is no order required or suggested, it will probably become another coding style battle.

Do not forget that static and are two different things. The keyword static relates to the lifetime, but relates to whether the expression can/must be executed at compile time.

If you currently mark functions as to denote they are not using any (non-static) data members, then you can (most likely) continue to do so, and mark them with constexpr or consteval as they need it.

When you mark a variable with that implies it is also Just to mention that there is such a thing as constexpr const in a situation where you want a compile-time constant expression of a pointer to a constant.

Recipe 7.4: constinit

Problem: We want a mechanism to ensure a variable has static initialization.

Solution: Also from C++ 20, we have the specifier

Another const-related keyword to talk about is which asserts a variable has:

Zero Sets the value of an object initially to zero.

Constant Sets the values of the static variables to a compile-time constant.

Done in that order, after all global variables are zero-initialized, constant initialization takes place. A constinit value must be set with a constant expression.

The tricky part about a specifier that seems to stand for constant initialization is that it means it must be initialized with a constant at compile time, and the variable itself is not constant at runtime. If we make another change to our previous example:

```
{
```

```
people) {

return people + (people /

    }

num)

    {

#ifdef __GNUC__


#else


return == num)

            ?

            : + ((num > :

#endif

    }

{
```

```
guests =

household =

unexpected =

probablyPotatoCount =
HowManyPotatoesRequiredForMashed(household + guests);

maybePotatoCount = HowManyPotatoesRequiredForMashed(household +
guests + unexpected);

Chapter7_4_Average =

#ifdef __clang__

potatoCount = (probablyPotatoCount + maybePotatoCount) /
Chapter7_4_Average;

#else

potatoCount = + maybePotatoCount) / Chapter7_4_Average);

==

#endif
```

```
    }

{

count = LookAtPotatoCount();

        count *=

return count;

    }


{

<< << << __FUNCTION__ << ", for " << COMPILER <<

#ifdef __clang__

<< "only partially implemented" <<

#endif

auto number = GetPotatoCountThatIncludesLeftovers();
```

```
<< "The potato count is: " << number <<

    }

};
```

We want to get the value that we had before, which was calculated at compile time. Now, using we have ensured our variable was set/initialized at compile time. As you can see, our potato count is constant, and the goal is to cook enough potatoes that we will have plenty of leftovers the next day, since the result is 18, we should have more than enough.

The specifier constinit will probably not be used as much as the other new terms. There will be some cases, such as using a mutex, where it will be handy. A good usage is for when you want to avoid a race condition, otherwise known as a static initialization order fiasco

Now that we have discussed the various const-related keywords, you might be asking: Which one do I The answer of course, depends on why and when. Although not definitive, here is a chart that can help guide you. Refer to Figure

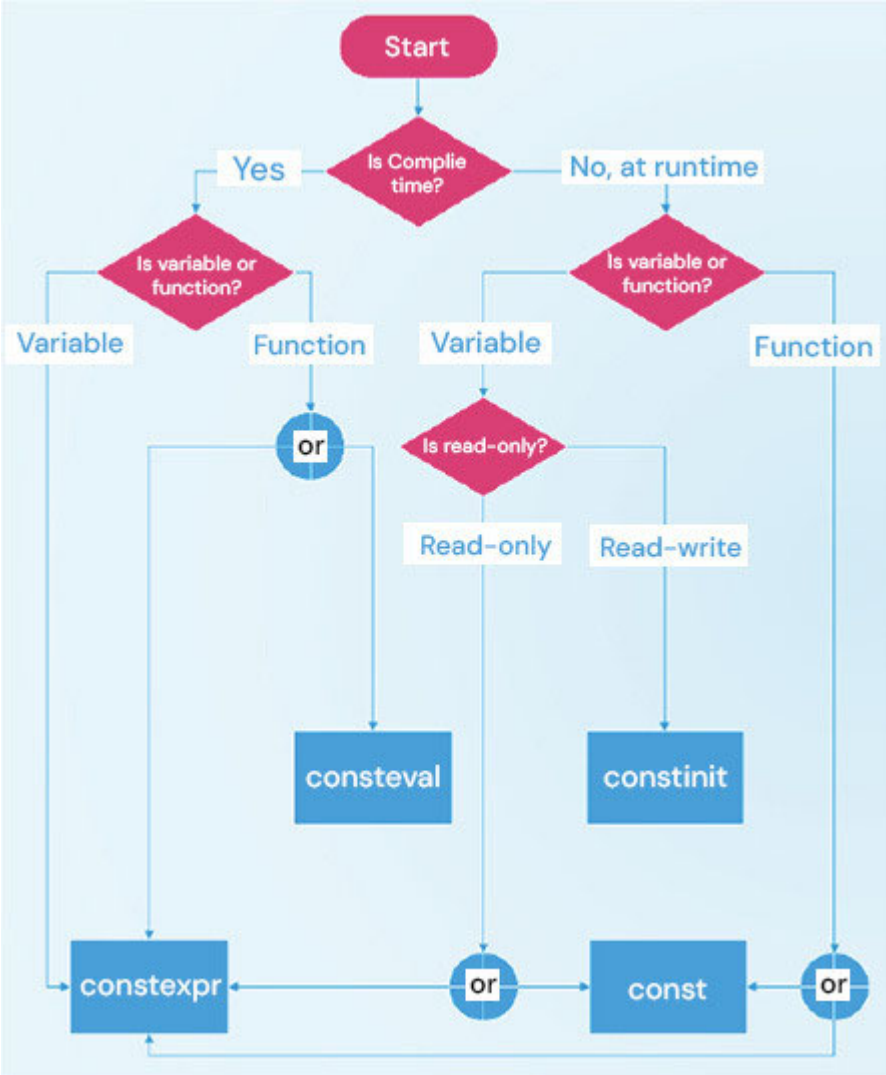

Figure 7.5: Flow chart of which keyword to use

Recipe 7.5: constexpr if

Problem: We want a mechanism to determine if the code we are running, is in a constant expression condition.

Solution: For C++17, the constexpr if was added.

There were some hurdles for the constexpr keyword to gain wide acceptance. One was that the standard library had too much functionality that did not use it. This meant either trying to do tricky workarounds or using third-party libraries. Using other libraries was fine, except that meant (possibly) a bigger code base, more external dependencies, more testing, and other concerns. Another issue was that was difficult to check on values. A normal if statement is executed at run-time, but since we have some constant expressions that are executed at compile time, there is no mechanism for this. C++17 came out with a way to check that. If we add a bit more code to our previous sample, as follows:

```
{



people) {

return people + (people /

    }
```

```
num)

    {

#ifdef __GNUC__


#else


return == num)

            ?

            : + ((num > :

#endif

    }

{

guests =

household =

unexpected =
```

```
probablyPotatoCount = HowManyPotatoesRequiredForMashed(household +
guests);

maybePotatoCount = HowManyPotatoesRequiredForMashed(household +
guests + unexpected);

Chapter7_5_Average =

#ifdef __clang__

potatoCount = (probablyPotatoCount + maybePotatoCount) /
Chapter7_5_Average;

#else

potatoCount = + maybePotatoCount) / Chapter7_5_Average);

==

#endif


    }


{

Increaser =
```

```
count = LookAtPotatoCount() * Increaser;

return count;

    }

Chapter7_5_1_UpperLimit =

{

#if (__GNUC__) && >= 201606L)

< Chapter7_5_1_UpperLimit) {


        }

else {

// return someUnknownValue;

// return "wow, that is a lot!";


        }

#else

// Not working yet with compiler
```

```
#endif
    }

{

<< << << __FUNCTION__ << ", for " << COMPILER <<
#ifndef __GNUC__
<< "only partially implemented" <<
#endif
auto decision = BuyBigBagOfPotatoes();
<< "The idea of buying a big bag is: " << << decision <<
    }
};
```

You will see that at compile-time, we are checking the constant expression result, and picking a code execution path. You can base it on type, or type size, or whatever you would know at compiler execution time. Note that in

our else block, we returned a string instead of a Boolean, which the compiler is fine with since we are returning an auto. What the compiler did not like, is when we tried to return someUnknownValue (since we have not declared that variable). Unlike an #if / #else block, all code must compile, regardless of whether it will be used.

The constexpr if did help out in many cases as you did not have to use messy template logic, using or other workarounds.

To help you understand, we will show you another scenario, the following code will not compile: (See references)

```
{



// will not compile !

T>



T& value) noexcept {

if {

return value;

        }

else {
```

```
        }

    }

{

<< << << __FUNCTION__ << ", for " << COMPILER <<

<< "does not compile properly" <<

int amount =

=

//std::cout << "Serving size of " << amount << " is " <<
GetServingSize(amount)

//    << ", and size of " << servingSize << " is " <<
GetServingSize(servingSize) << std::endl;

    }

};
```

Our templated function GetServingSize can look at different types. If it is an integer, it will simply return that value, otherwise, it will call the size() function on that type. Our code will try to call it twice, once for an int, and once for a string. The compiler will fail though, saying it does not know how to execute the size() function on an integer. The compiler tries to evaluate all of the code, based on the given type, and in the first example, it was an integer, which caused the failure. If we modify the function to use a constexpr

```
{



T>

T& value) noexcept {

{

return value;

  }

else {


  }

}

T>
```

```
T& value) noexcept {

val) {

{ val.count() } ->

  }) {



  }

val) {

{ val.size() } -> })

{



  }

val) {



{ val } -> })

  {
```

```
return value;

  }

else {


  }

}


{

<< << << __FUNCTION__ << ", for " << COMPILER <<

int amount =

=

<< "Serving size of " << amount << " is " << GetServingSize(amount)

  << ", and size of " << servingSize << " is " << GetServingSize(servingSize)
<<

biggerServiceSize{ 's' };

biggerServiceSizeText = biggerServiceSize.end(),
```

```
<< "Or serving size of " << amount << " is " <<
AnotherGetServingSize(amount) <<

  << ", and size of " << servingSize << " is " <<
AnotherGetServingSize(servingSize) <<



  << ", and size of " << biggerServiceSizeText << " is " <<
AnotherGetServingSize(biggerServiceSize) <<

}

};
```

The compiler will evaluate the GetServingSize function, and it will know that there are two different types being used. It knows to generate two overloads of the function, one for int, and one for string.

We also created the function which does not use a type trait as GetServingSize does. This uses a small concept, where we define what needs to be acceptable for that block of code, and if acceptable it will execute the code for that requirement. This method is a bit more intricate, but if your needs are that you need that level of detail then it certainly is a good feature to use. Like other or switch blocks, the order does matter, since it will execute the first acceptable requirement.

The output can be seen in Figure

```
Chapter7_5_3::GetServingSizeDescription, for MSVC: 1939
Serving size of 42 is 42, and size of lots is 4
Or serving size of 42 is 42
, and size of lots is 4
, and size of thousands is 9
```

Figure 7.6: Output for GetServingSizeDescription

Recipe 7.6: if consteval

Problem: We want a mechanism to determine which code path to take, depending on if we are executing code at compile-time.

Solution: In C++23, we have if

The if constexpr (or constexpr if as it is named) addition helped many people, but there was also a need to check if the code being executed is within a constant-evaluated context. In C++20, they added which does not take any parameters and returns a Boolean. It asks whether this is being executed in a constant expression context. Adding more to our previous example, we have:

```
{



people) {

return people + (people /

}

num)

    {
```

```
#ifdef __GNUC__


#else

return == num)

        ?

        : + ((num > :

#endif


    }

{

guests =

household =

unexpected =

probablyPotatoCount =
HowManyPotatoesRequiredForMashed(household + guests);
```

```
maybePotatoCount = HowManyPotatoesRequiredForMashed(household +
guests + unexpected);

Chapter7_5_Average =

#ifdef __clang__

potatoCount = (probablyPotatoCount + maybePotatoCount) /
Chapter7_5_Average;

#else

potatoCount = + maybePotatoCount) / Chapter7_5_Average);

==

#endif


    }


Chapter7_6_1_UpperLimit =

{

#ifdef __GNUC__
```

```
< Chapter7_6_1_UpperLimit) {



        }

else {



        }

#else

// Not working yet with compiler



#endif

    }

{

count = LookAtPotatoCount();

        count *=
```

```
return count;

    }

{

if {

auto count = LookAtPotatoCount();

return count;

        }

else {


        }

    }

{

<< << << __FUNCTION__ << ", for " << COMPILER <<


#ifndef __GNUC__
```

```
<< "not implemented yet" <<

#endif

int result = GetPotatoCountThatIncludesLeftovers();

count_constexpr = AskHowManyPotatoesToBuy();

= AskHowManyPotatoesToBuy();

= AskHowManyPotatoesToBuy();

<< " count_constexpr=" << count_constexpr <<

"\n count_auto=" << count_auto <<

"\n count_const=" << count_const

            <<

return result;

    }

};
```

Our new function says that if we are running in a constant context, then return the count, otherwise just return 100.

We then call it 3 times setting variables that are auto, const, and as shown in Figure

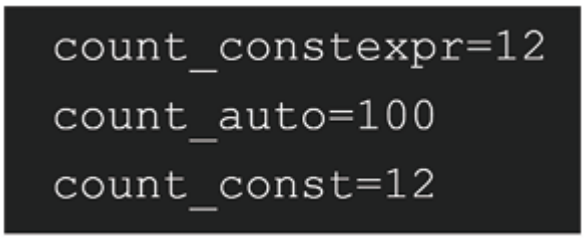

Figure 7.7: Output of is_constant_evaluated

You can see in the output, that when our variable is const or we get the result of the if block. Otherwise, when not in a constant context, we get our default value of 100.

The function is_constant_evaluated is handy when you might have two different code paths you want to run. When is_constant_evaluated() is true, you might run some different code path, since it cannot (easily) do dynamic memory, or throw exceptions.

The tricky thing here might be if you want to check if you are executing in compile time. In this case, is_constant_evaluated will fail if you run it within a constant expression. In some compiler scenarios, code such as the following would give a compiler error:

It says in a compile-time context, you are checking if we are running at compile time, which does not make sense. Plus is_constant_evaluated has had other problems, when using non-integral types.

In C++23, they came out with another check, named consteval Although it too has some drawbacks, it simplified some shortcomings of If we redo our method, as follows:

```
{



people) {

return people + (people /

    }

num)

    {

#ifdef _MSC_VER

return == num)

            ?
```

: + ((num > :

#else

#endif

}

{

guests =

household =

unexpected =

probablyPotatoCount = HowManyPotatoesRequiredForMashed(household + guests);

maybePotatoCount = HowManyPotatoesRequiredForMashed(household + guests + unexpected);

Chapter7_6_Average =

#ifdef __clang__

```
potatoCount = (probablyPotatoCount + maybePotatoCount) /
Chapter7_6_Average;

#else

potatoCount = + maybePotatoCount) / Chapter7_6_Average);

==

#endif



    }

Chapter7_6_2_UpperLimit =

{ }

{

#ifdef __GNUC__

count = LookAtPotatoCount();

(count < GetLimit()) {



        }
```

```
else {

        }

#else

// Not working yet with compiler


#endif

    }

{

#if >= 202106L

{

        }

else {
```

```
        }
#else

#endif
    }
{
count = LookAtPotatoCount();
        count *=
return count;
    }

{
<< << << __FUNCTION__ << ", for " << COMPILER <<
#ifndef __GNUC__
```

```
<< "not implemented yet" <<

#endif


int result = GetPotatoCountThatIncludesLeftovers();

count_constexpr = AskHowManyPotatoesToBuy();

<< "count_constexpr=" << count_constexpr <<

= AskHowManyPotatoesToBuy();

<< "count_auto=" << count_auto <<

= AskHowManyPotatoesToBuy();

<< "count_const=" << count_auto <<

return result;

}

};
```

You can see in our AskHowManyPotatoesToBuy() function, that the consteval if looks different. It is the only control-flow construct that does not have parentheses, and unlike other if statements, it requires braces to

define the if (and block. If you do not comply, it will give you a compiler error. If consteval does a similar purpose; it checks if a constant expression is being evaluated in a constant-evaluated context.

Recipe 7.7: Some hidden problems with constexpr

Problem: Although these new const-related keywords are handy, there are still ways to get into trouble.

Solution: Nothing magical, one just has to be aware of the issues.

Not related to any modern keywords, but just remember the positioning when you are looking at const code. Here is a code sample, that could have been taken from many releases ago, but still causes the occasional problem.

```
{



{



Name;

    };



{
```

```
<< << << __FUNCTION__ << ", for " << COMPILER <<

        Recipe r;

        Recipe* pr = &r;

const Recipe* otherR = &r;

//otherR->Name = "something great!"; // cannot change content

        Recipe* = &r;

Recipe r2;

//pFixedR = &r2; // cannot change where it is pointing

    }


};
```

We have two compiler issues:

otherR is a pointer to a const object, and so the content values cannot be changed.

pFixedR is a constant pointer to an object, so the content can change, but what it points to, cannot.

But for modern const issues, constexpr has had its share. As mentioned earlier, constexpr says that it must be able to execute at compile-time, but unlike constexpr may not run at compile-time. We have already mentioned the statement, but to get around this, developers have used which runs at compile time. If you have a constexpr result that you want at compile time, one way to ensure that, is to check that result with If you just used the normal that is executed at runtime, your issue will not be resolved.

If your goal is to run at compile time, then a static_assert is recommended. This is because, on some compilers, it may not give you proper warnings or errors. If you have, near the end of your call chain, forgot to state then it will just fall back to being run-time execution. Unless you are checking the assembly instruction output, you may not notice it did not create compile-time execution code, and so static_assert not only forces the compiler to do what you wanted the compiler to do, but it makes the compiler check your code to ensure it was done how the compiler needs it to be.

There are some cases where a constexpr variable is not doing exactly what you expected. To resolve that, you can use static constexpr to ensure it is static.

Declaring a static constexpr variable in a header may also give you an undesired result. It could instantiate that (potentially large) object in every code file it is declared in. You may have wanted to declare it as inline constexpr instead. Another issue one might have relates to One Definition Rule Shortcomings This can sometimes happen with statics and relates to something being declared, but not fully defined.

Even with C++23, there are shortcomings with in that you might have a compile-time result. People have been doing some coding tricks to work around that. Hopefully, the language capabilities improve, but if you are in one of those situations, there are workarounds you can try.

Unrelated to the other rules, it is highly advised not to use const_cast to remove the const-ness of an object. It is a strong code-smell of something not designed properly and creates a very big chance of something not working the way you hope it to. Use it with extreme caution.

Conclusion

The capabilities of the compilers have grown substantially in recent years, but to ensure that ability, you may have to alter your code to empower it. Some of these code changes are not big, nor harsh. Using the new const-related keywords gives your code a new ability, and most likely with less effort than doing a lot of big template metaprogramming. If you were trying to achieve those goals before, then you should be able to do so with less code and it will look cleaner. The more work your compiler can do, the less work the program has to do at run-time.

Points to remember

Executing code at compiler time, versus at run time, should make the execution of your program faster.

There are new const-related keywords; understand their differences to see what you should be using in different situations.

You should do enough to verify your compile-time code is doing what you hoped it was doing. You should use asserts and other checks, but remember that you are limited in what you can do if there is an issue.

Consider using static since if a method's goal is to provide a value, and can do so at compile-time, then you might be able to reduce creating new instances of variables by being static.

References

CPPreference. (n.d.).

IsoCpp. (n.d.).

StackOverflow. (n.d.).

StackOverflow. (n.d.).

CppReference. (n.d.). Retrieved from https://en.cppreference.com/w/cpp/language/definition

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 8

Concurrent Processing

Introduction

Naturally, as the world evolves, things get more complicated and we want things done as soon as possible. This, in the software world, means that we want them to run concurrently, although concurrent does not guarantee faster; it just means organizing things so some can get the focus they need and others can wait, which often means the net result is faster. C++ 23 continues the trend of providing more support so that developers can make their code smarter in terms of dividing up the work to complete it more efficiently.

Having an application that works concurrently is no longer nice to have but essential. Distributed systems and web services running in the background to provide updated information, means that today's users need to have an application that responds to their actions, and provides information back to them as soon as it is available.

For such an important part of coding, the concurrent library has had important and welcome enhancements, although no sweeping changes in recent releases. There were a few features that people were hoping might get into C++23 that did not, but perhaps were a testament to a good initial design.

Structure

In this chapter, we will cover the following recipes:

Recipe 8.1: std::jthread is the new thread

Recipe 8.2: std::stop_token

Recipe 8.3: std::stop_callback

Recipe 8.4: std::stop_source

Recipe 8.5: std::counting_semaphore, and std::binary_semaphore

Recipe 8.6: std::atomic, and std::atomic

Recipe 8.7: std::atomic::wait, and std::atomic::notify*

Recipe 8.8: std::latch

Recipe 8.9: std::barrier

Recipe 8.10: std::atomic_ref

The good news is that all three compiler makers have done a good job implementing the features we are about to discuss:

```
}
```

Objectives

Computers have become a lot faster; systems have become more complex, and users demand results sooner. There are many reasons why concurrency is becoming an essential and integral part of programming. By the end of this chapter, the reader will understand more about how concurrent code is done in C++, and how to use some of the latest standard library additions.

Recipe 8.1: std::jthread is the new thread

Problem: We need a thread that automatically rejoins on destruction.

Solution: At long last, C++ 20 finally has

To start of the example, we will include the headers we will use in this chapter:

```
#include

#include

#include

#include

#include

#include

#include

#include
```

```
#include

#include

#include

#include

#include

#include

#include

#include

#include

#include
```

We will also show here a utility class for output, that we will use throughout the chapter:

```
{




output) {
```

```
scNow =

ms = %

=

tmNow;

#ifdef _MSC_VER

localtime_s(&tmNow,

#else

&tmNow);

#endif



oss <<

oss << << << << ms.count();

<< oss.str() << ": " << output <<

}

message, int threadNumber) {
```

```
oss << "Thread counter=" << threadNumber;

oss << ", " << message;



}

};
```

The addition of std::thread in C++11 was a great addition. It allowed you to run code in the background on another thread.

```
#include "Chapter8.h"

#include

#include

{

// can do a lot of other work in this spawned thread

}

{
```

```
// critical not to forget this

// myThread.join();

}


{

CallThread();

}
```

But soon after std::thread was released, people noticed that there was a serious design flaw. If you did not join the thread back before the thread ends, then it could easily crash the program. It was not long afterwards that people had proposals for fixes, but it was not accepted. More years went on, and more proposals came, but it was still not accepted. The solution for many was to switch to a third-party library or create a wrapper to improve stability. Finally, a compromise was made in C++20; instead of fixing the old class, they would create a new one. The new std::jthread works like the old but without the instability. You could just do a find-and-replace on the name, but the new class does come with enhancements. The J of jthread is for joinable, meaning that it calls join on the thread in the destructor, to keep the program running smoothly.

At the very least, using jthread is safer, since jthread will automatically join back once it goes out of scope:

```
{


{

// can do a lot of other work in this spawned thread

}


{


// it will not longer crash if you forget this

// but you should still call it

// myThread.join();

}

};
```

Note: Just because jthread is safer, in that it will automatically join so the application will not crash, does not mean that you should always ignore when to properly join back and how to manage your threads.

We will go back to our editor to make new code that does something similar, just so we can build off of it.

```
{



{

 cooking



 to

}

{

<< << << __FUNCTION__ <<



}

};
```

Adding in a bit of code relating to time, will let us see the results easier. We can otherwise ignore the function since it is not the nicest to look at. Our code in makeTheMeal will output parts of a message, with a sleep_for in the middle. Refer to Figure

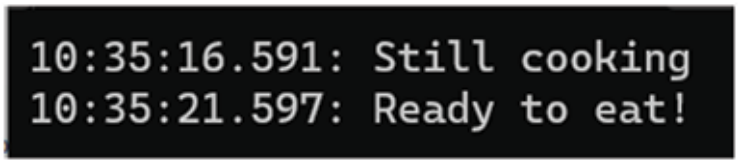

Figure 8.1: output from makeTheMeal

Keep in mind that sleep_for is not related to the jthread class, and it works on whatever the current thread is. We have a sleep_for in our output function, and our calling function, both delaying different threads. We did not neatly join our thread back, but at least with that did not cause our application to terminate.

As mentioned before, other than it automatically joining (if required), std::jthread has similar behavior to its counterpart:

Execution starts automatically upon the constructor, once complete.

If the callable throws an exception that is not handled, then std::terminate is called.

The callable function can be void since it does not return any value back to where the thread is defined. If a return value is wanted, then you would use However, if we did use we could just return the value, but we will get into all of that soon enough.

Since we just mentioned it, we can go over how a thread can send and retrieve data. Once again with C++, there are a few different ways to do something. Some are easier than others, and some might be better, depending on the particular situation. If we try making a few dishes in our meal:

```
{



food) {

  the " + food);

  food = "delicious " + food;

}

food) {

  the " + food);

  food = "tasty " + food;

}



food) {

  the " + food);
```

```
  food = "yummy " + food;

}

{

<< << << __FUNCTION__ <<

=

foodToCook);

  the " + foodToCook);

// will not compile

foodToCook);

  the " + foodToCook);
```

```
  the " + foodToCook);

}


};
```

Now we are trying to create three different threads, and in each thread, we modify the food that is passed in, and for each child thread, we join and print out the variable that we passed into the thread.

Unfortunately, this code does not even compile. In the middle attempt, when we call we pass the variable, but we get a compiler error saying you cannot pass by value with a thread. It will always pass by a copy, unless we use as we did in the third scenario. If we get rid of the bad code, as follows:

```
{

=

foodToCook);

cookingThread1.join();

the " + foodToCook);

//std::jthread cookingThread2(cookTheDishByReferenceMaybe,
foodToCook);
```

```
//cookingThread2.join();

//outputText("made the " + foodToCook);


cookingThread3.join();

the " + foodToCook);

}
```

Then we can run the program and see our output as shown in Figure

```
11:41:34.448: cooking the lasagna
11:41:34.452: made the lasagna
11:41:34.453: cooking the lasagna
11:41:34.454: made the yummy lasagna
```

Figure 8.2: Output from makeAnotherMeal

We can see that for the first attempt in we pass the food parameter by value, and so naturally, changing that value will not affect the variable that was passed in. In the next attempt, we use std::ref to pass to The thread calls a function. When we print the food that we had passed in, once that function has been completed, we see that the value has changed.

This raises an important point: Although this code ran successfully, it is dangerous to just pass a reference to a thread, without using other

synchronization mechanisms. If we started to use a variable, but we also changed the variable's value in the parent thread, then we could easily have unwanted behavior.

While on the topic, threads (or are also not allowed to be copied, or assigned.

foodToCook);

// will not compile, since we cannot use copy/assignment operations with threads

//std::jthread cookingThread1b = cookingThread1;

cookingThread1b =

//cookingThread1.join(); // no longer using this thread

cookingThread1b.join();

The code sample shows that we would get a compile error if we simply tried to assign the thread, so that too needs a std::move to transfer ownership.

Yet another way of having data interacting with your thread is using std::future and It means you are giving the thread a promise that you will provide some data in the future. The templated types are the preferred way by many, but since we are being ambitious in showing so many different mechanisms, the other type to talk about is Take a look at an example of and std::async in the following code:

```
{



foodFuture) {

  about


auto food = foodFuture.get();

  " + food);

if == {

is impossible to

  }

  to


}


food) {

  the " + food);
```

```
  food = "delicious " + food;
}

{
<< << << __FUNCTION__ <<
  passing by
cookTheDishByValue,
  by promise right
prom1;
fut1 = prom1.get_future();


result1 = cookTheDishByFuture,

  of cooking was " + ? "good" :
  by promise, but making thread
prom2;
```

```
fut2 = prom2.get_future();

result2 = cookTheDishByFuture,




 of cooking was " + ? "good" :

}

};
```

async is another templated type introduced in C++11, and is used by many because it was an easier transition using their existing code to make it into something asynchronous. If you look at our makeMealInTheFuture function, the first time we called std::async was with our existing cookTheDishByValue function, which worked fine. One other point about is that you should not await your calls in the destructor. It does not seem like the best place and could lead to issues. It does not make sense for your object to end before your task is finished, so if that is the case, then it could be a design issue.

Most meals were good, or at least, all went as planned, as shown in Figure

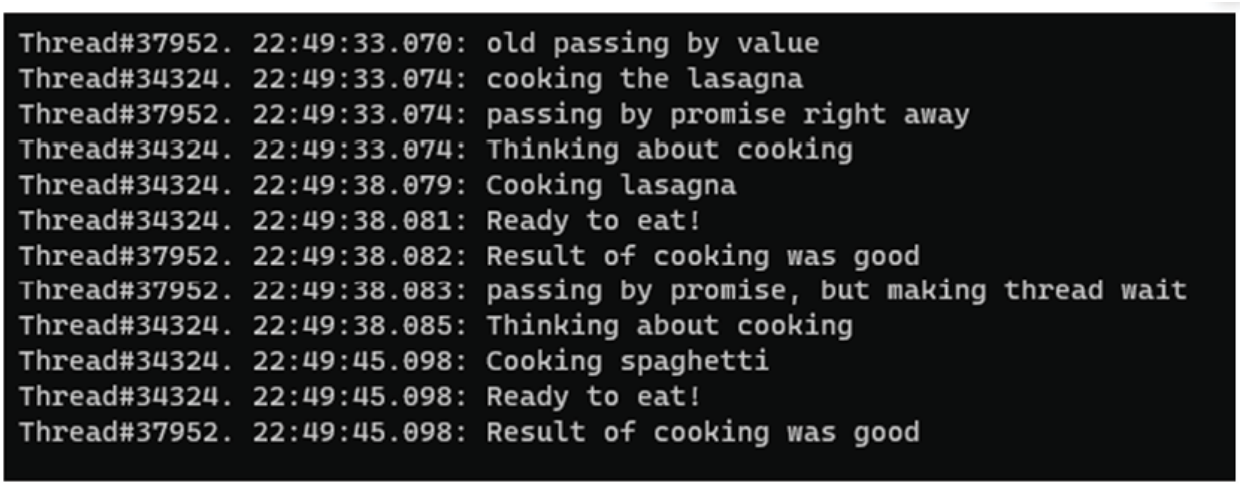
Thread#37952. 22:49:33.070: old passing by value
Thread#34324. 22:49:33.074: cooking the lasagna
Thread#37952. 22:49:33.074: passing by promise right away
Thread#34324. 22:49:33.074: Thinking about cooking
Thread#34324. 22:49:38.079: Cooking lasagna
Thread#34324. 22:49:38.081: Ready to eat!
Thread#37952. 22:49:38.082: Result of cooking was good
Thread#37952. 22:49:38.083: passing by promise, but making thread wait
Thread#34324. 22:49:38.085: Thinking about cooking
Thread#34324. 22:49:45.098: Cooking spaghetti
Thread#34324. 22:49:45.098: Ready to eat!
Thread#37952. 22:49:45.098: Result of cooking was good

Figure 8.3: Output from makeMealInTheFuture

In the next case, we create a std::promise and set the value. Then, we create a std::future with that We then call std::async using that future (passed by reference). In our calling function, we get the future's value with the get() function, and cook that food and (hopefully) say the meal was a success. Note the future.get() in our calling function will block and wait for a value to use, because that is what it was promised. The get() function effectively calls the wait() function, which waits for the result but does not return anything. The other couple of wait-related functions for a future are which waits for the result up to a specified duration, and which waits for the result up until a specified time.

In our third scenario in we changed the order to prove the delay. We call while keeping in mind that the function cookTheDishByFuture will start executing right away. In the output, we can see it is in the thread we created, but once it tries to get at that passed-in value, it has to wait a bit to get it.

As you can see, we are explicitly passing in a future string, and we get back a future of type Boolean. In our first scenario with we do not set the std::async result to a variable, because the function was originally void. For the others,

we made sure we captured the result. This is because, since C++20, the result of std::async was adorned with the nodiscard attribute, which means the value should be looked at, and not just thrown away. That makes sense, considering if we call a method that returns a result, then we should use it.

We will finish off this diversion looking at promise by looking at exception handling, as follows:

```
{



foodFuture) {

  about


auto food = foodFuture.get();

  " + food);

if == {

is impossible to

  }

  to
```

```
}

{
<< << << __FUNCTION__ <<
try {
prom3;
fut3 = prom3.get_future();
  by promise something extremely difficult to


result3 = cookTheDishByFuture,


  of cooking was " + ? "good" :
  }
catch ex) {
```

```
  of cooking was... really



  }

try {

  by promise, but then breaking

prom5;

fut5 = prom5.get_future();

result5 = cookTheDishByFuture,


  out of food to

  of cooking was " + ? "good" :

  }

catch ex) {

  of cooking was... really
```

```
  }

}

};
```

In the first scenario, it is similar to before, so all should work well, but we passed in a value requesting a meal of Filindeu (we do not know what it is either; we just searched for impossible pasta to make and that is what we found), and our function says that it cannot make that so it throws an exception. Our calling function catches the exception and handles it nicely.

We also said that if we fail to deliver on our promise, then it will throw an exception. Sometimes though, we might start a thread, only to realize we do not have any data to give to it. To terminate it semi-gracefully, we can have the promise set an exception, which will end that thread and return to the code that called it. Both those cases assume you have done proper exception handling where you started the new thread. In the second scenario above, we called set_exception on our promise, which breaks out of our thread, and we can catch that exception to continue onwards.

There is something to note about the preceding code. You noticed with we passed in the first parameter of That means that it has to start execution on the new thread right away. The other option is which means execution will not start until we ask for the result.

There are a few more older types for C++ concurrency, such as and std::mutex that we will work on later. For now, we go back to looking at newer classes, and how they relate to

Recipe 8.2: std::stop_token

Problem: We need a mechanism to check if the thread has been asked to stop.

Solution: In C++ 20, along with they have the class

We saw in the previous topic, that if we send an exception to a promise, that would stop the execution of a thread. There is another constructor for that automatically passes a which is a way to check if a stop has been requested. Then on your thread, you can simply issue a and it will get triggered in your function running on the other thread. We created another example here:

```
{


start,

end) {


diff = end - start;

  oss << "duration of " << << << diff.count();


}
```

```
stoptk, food) {

start =



  to cook the " + food);

for = i < i++) {

if {

auto durationText = getDurationText(start,

    cooking the " + food + " early, after " + durationText);



  }







  }

auto durationText = getDurationText(start,

  time to stop cooking the " + food + ", " + durationText);
```

```
}

{
<< << << __FUNCTION__ <<

  cookingThread1.join();




  cookingThread2.join();


}
};
```

You will notice that in it has a stop_token parameter, which we did not have to explicitly pass in. Our normal parameters are just added to the end of the function. In our function, we create a loop (where we imagine we were busy cooking). In that loop, you will see if stop_requested() was triggered, and if so, then we can clean up and exit out cleanly.

Before we look at what we do with the threads, let us look at the output shown in Figure

```
Thread#31136. 16:23:15.435: starting to cook the lasagna
Thread#31136. 16:23:15.437: still cooking
Thread#31136. 16:23:15.546: still cooking
Thread#31136. 16:23:15.655: still cooking
Thread#31136. 16:23:15.764: still cooking
Thread#31136. 16:23:15.871: still cooking
Thread#31136. 16:23:15.982: still cooking
Thread#31136. 16:23:16.091: still cooking
Thread#31136. 16:23:16.201: still cooking
Thread#31136. 16:23:16.309: still cooking
Thread#31136. 16:23:16.419: still cooking
Thread#31136. 16:23:16.527: regular time to stop cooking the lasagna, duration of 1.0921
Thread#33196. 16:23:16.529: starting to cook the pizza
Thread#33196. 16:23:16.530: still cooking
Thread#33196. 16:23:16.636: still cooking
Thread#33196. 16:23:16.745: still cooking
Thread#33196. 16:23:16.854: still cooking
Thread#33196. 16:23:16.964: stopped cooking the pizza early, after duration of 0.434042
Thread#25912. 16:23:16.966: starting to cook the soup
Thread#25912. 16:23:16.966: still cooking
Thread#25912. 16:23:17.074: still cooking
Thread#25912. 16:23:17.182: stopped cooking the soup early, after duration of 0.216119
```

Figure 8.4: Output of stopTheMeal

In the first scenario, we start our and then call join. Remember calling join just means it will wait for our thread to finish to its normal completion. We can see that the output says it got to the regular time to stop cooking after about a second (10 iterations of a loop that sleeps for 100 milliseconds).

For the second scenario, we start our thread, and then the main thread sleeps for 400ms (or 0.4 seconds as the output shows). We then issue a request_stop on our thread, and the output says it stopped early, after four iterations (which happens to be four times a 100ms sleep).

The final scenario has our thread starting, we sleep for 200 ms, and two iterations of our loop is all that we got, but why did it stop since we did not ask it to? The reason for that, is that we did not call join on our child thread, and our function kept running until it completed. Once our main function ended, the jthread becomes out of scope and its destructor is called. In destructor, it calls in hopes of stopping the thread cleanly.

There can be situations where you would want an immediate termination of the thread, and we have seen this can be accommodated when you give an exception to your promise. For terminations that are more controlled, using stop_token seems a more usable approach.

Recipe 8.3: std::stop_callback

Problem: We would like a mechanism for a callback function for our thread.

Solution: Another jthread related type for C++20 was

In our last section, we learned how to use a stop_token to gracefully exit out of our function running in another thread. Another feature that was added to jthread's cooperative cancellation methodology was the addition of a callback mechanism. In the child function that is passed a you can define (or register, if you prefer that term) a callback using the existing The sample code we have uses a function similar to what we previously used, and follows a similar approach. We create a thread providing a function to execute. In that function, we have a loop, in which we check if stop_requested was set:

```
{



start,

end) {



diff = end - start;

  oss << "duration of " << << << diff.count();
```

```
}

stoptk, food) {


start =

  to cook the " + food);

for = i < i++) {

if {

auto durationText = getDurationText(start,

    cooking the " + food + " early, after " + durationText);


  }




  }

auto durationText = getDurationText(start,
```

```
  time to stop cooking the " + food + ", " + durationText);

}

{

  callback

}

stoptk, food) {

start =

  to cook the " + food);


cb1{ stoptk, [] {

  to

  } };

cb2{ stoptk, callbackFn };

for = i < i++) {

if {
```

```
auto durationText = getDurationText(start,
    cooking the " + food + " early, after " + durationText);
  }
  }
auto durationText = getDurationText(start,
  time to stop cooking the " + food + ", " + durationText);
}
{
<< << << __FUNCTION__ <<
  a thread to cook
```

```
  cookingThread2.join();

}

};
```

The main addition this time is in our function, we use the stop_token that was provided to us and register a callback. In our example, we registered two callbacks; one is a little lambda, and the other is a function, as shown in the following Figure

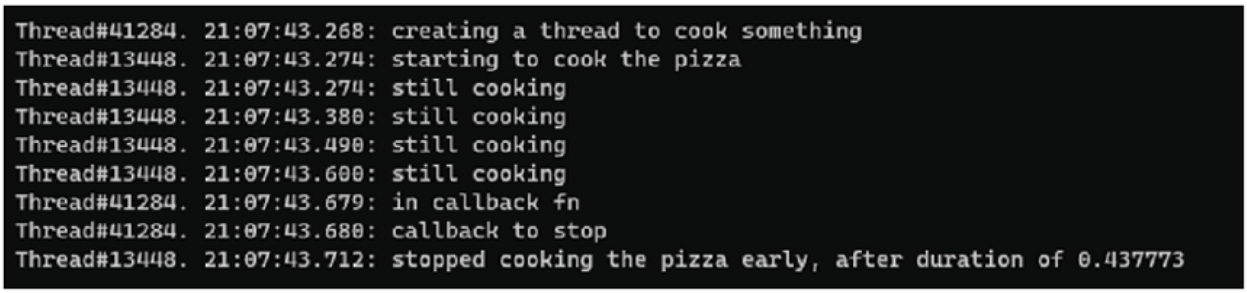
Thread#41284. 21:07:43.268: creating a thread to cook something
Thread#13448. 21:07:43.274: starting to cook the pizza
Thread#13448. 21:07:43.274: still cooking
Thread#13448. 21:07:43.380: still cooking
Thread#13448. 21:07:43.490: still cooking
Thread#13448. 21:07:43.600: still cooking
Thread#41284. 21:07:43.679: in callback fn
Thread#41284. 21:07:43.680: callback to stop
Thread#13448. 21:07:43.712: stopped cooking the pizza early, after duration of 0.437773

Figure 8.5: Output from stopTheMealWithCallback

Looking at our output, we first print a line from our main thread and then our familiar output from the child thread, which we issue a request_stop on. The output shows us that both of our callbacks were called. You can register multiple callbacks for a single but the order in which the callables are executed, is not defined.

Note: You will see our main thread ID 41284, and then when we were cooking in the function being run on another thread, it was using thread ID 13448. Note though, in the output of our two callbacks, that they are both

using thread ID 41284. The callbacks are executed in the thread that created the thread.

Keep in mind for our simple example, that we are just outputting some text. Most likely in your scenario, you will be working with some data, in which case you should protect it with a mutex or some other concurrency mechanism so that there are not any problems.

Recipe 8.4: std::stop_source

Problem: We want a way that we can stop multiple child threads with one request.

Solution: std::jthread also interacts with the new class

The next topic relates to the question of synchronizing multiple child threads, and how to stop them at the same time (or close to it) easily. The class std::stop_source is passed into the child function that will run in another thread, this time at the end of the parameter list. With the same stopper being passed in, we have a way of triggering a stoppage with one call. The request_stop is not called on the jthread as we previously did but on the stop_source object. Since the stop_source was passed into the jthread constructor, it knows which threads were created, and so it knows what threads to request to stop.

```
{



start,

end) {



diff = end - start;
```

```
  oss << "duration of " << << << diff.count();



}

{

  callback



}

food, stopper) {

start =

  to cook the " + food);

= stopper.get_token();

cb1{ stoptk, [] {

  to

  } };

cb2{ stoptk, callbackFn };

for = i < i++) {
```

```
if == 3 && food == {

stopper.request_stop();

  }

if {

auto durationText = getDurationText(start,

    cooking the " + food + " early, after " + durationText);


  }




  }

auto durationText = getDurationText(start,


  time to stop cooking the " + food + ", " + durationText);

}
```

```
{

<< << << __FUNCTION__ <<

  a thread to cook something, and then stop

stopper1;

stopper1);



  cookingThread1.join();

  a thread to cook something, and then stop it from

stopper2;

stopper2);

  cookingThread2.join();

  multiple threads, and then stop them all with the same

=
```

```
stopper3;


for n = n < thread_count; n++) {


oss << "dish" << n;

  threads[n] = oss.str(), stopper3);

  }




}

};
```

We are again using code that should look a bit familiar, except we define the stop_source and pass it in on the jthread constructor. We can then call request_stop on it, and we can see the results in Figure

```
Thread#21888. 22:05:53.990: creating a thread to cook something, and then stop it
Thread#17968. 22:05:53.997: starting to cook the pizza
Thread#17968. 22:05:53.997: still cooking
Thread#17968. 22:05:54.103: still cooking
Thread#17968. 22:05:54.212: still cooking
Thread#17968. 22:05:54.321: still cooking
Thread#21888. 22:05:54.400: in callback fn
Thread#21888. 22:05:54.400: callback to stop
Thread#17968. 22:05:54.431: stopped cooking the pizza early, after duration of 0.433625
Thread#21888. 22:05:54.433: creating a thread to cook something, and then stop it from inside
Thread#8136. 22:05:54.436: starting to cook the ravioli
Thread#8136. 22:05:54.436: still cooking
Thread#8136. 22:05:54.540: still cooking
Thread#8136. 22:05:54.650: still cooking
Thread#8136. 22:05:54.758: in callback fn
Thread#8136. 22:05:54.758: callback to stop
Thread#8136. 22:05:54.758: stopped cooking the ravioli early, after duration of 0.322716
Thread#21888. 22:05:54.837: creating multiple threads, and then stop them all with the same stop_source
Thread#36220. 22:05:54.839: Thread#37632. 22:05:54.840: starting to cook the dish1
Thread#30100. 22:05:54.840: starting to cook the dish2
Thread#37632. 22:05:54.840Thread#30100. 22:05:54.840: still cooking
starting to cook the dish0: still cooking

Thread#36220. 22:05:54.840: still cooking
Thread#36220. 22:05:54.949: still cooking
Thread#30100. 22:05:54.949: still cooking
Thread#37632. 22:05:54.949: still cooking
Thread#37632. 22:05:55.059Thread#30100. 22:05:55.059: still cooking
: still cooking
Thread#36220. 22:05:55.059: still cooking
Thread#37632. 22:05:55.169: still cooking
Thread#30100. 22:05:55.169: still cooking
Thread#36220. 22:05:55.169: still cooking
Thread#21888. 22:05:55.248: in callback fn
Thread#21888. 22:05:55.249: callback to stop
Thread#21888. 22:05:55.249: in callback fn
Thread#21888. 22:05:55.249: callback to stop
Thread#21888. 22:05:55.250: in callback fn
Thread#21888. 22:05:55.250: callback to stop
Thread#30100. 22:05:55.279: stopped cooking the dish2 early, after duration of 0.439466
Thread#37632. 22:05:55.279: stopped cooking the dish1 early, after duration of 0.440009
Thread#36220. 22:05:55.279: stopped cooking the dish0 early, after duration of 0.44058
```

Figure 8.6: output from stopTheMealWithStopSource

In our cookTheDishButThenSourceStopWithCallback function that is running in another thread, we can see we can get a stop_token from the and we can register our callbacks as we did before.

The first two scenarios are similar, except that the first time we call request_stop from the parent thread, and in the second scenario we call request_stop from within the child thread.

In the last scenario, we create an array of threads and pass in the same Then, with a single call to we stopped all the child threads, although not in any

particular order.

Recipe 8.5: std::counting_semaphore, and std::binary_semaphore

Problem: We want an easy and lightweight way of controlling a shared resource.

Solution: In the C++ 20 release, there is and

The idea of semaphores has been around in software development for a very long time, but C++ did not have one built in. People either made a simple one based on a mutex, or they used a third-party library. C++ finally added one (or two, actually) in C++20, which was a welcome addition.

Before we get into the new let us go over a mutex example first, so that we can compare techniques:

```
mapMutex;

int>> recipeAndRating;

{


static void evaluateRecipe(std::string food, int secondsDelay) {

const int BestRating =
```

```
gen(rd());

// thinking about rating

std::this_thread::sleep_for(std::chrono::milliseconds(secondsDelay *

std::lock_guard guard(mapMutex);

std::pairint> pair(food, gen() % BestRating);


recipeAndRating.push_back(pair);

std::cout << "Recipe=" << pair.first

  << ", rating=" << pair.second <<

// guard goes out of scope, freeing up mutex

}


static void lookAtMutexMap() {

std::cout << std::endl << std::endl << __FUNCTION__ << std::endl;
```

```
gen (rd());

recipes[] = { "broiled "ravioli" };

std::jthread

  int threadCounter =

std::cout << "Looping through

for recipe : recipes) {

const int MaxSeconds =

  int secondsDelay = & MaxSeconds) +

std::cout << "Thinking about rating on " << recipe << " for " <<
secondsDelay << "

  threads[threadCounter] = std::jthread(evaluateRecipe, recipe,
secondsDelay);

threadCounter++;

  }

std::cout << "\nLooking at

while > {
```

```
threads[threadCounter -


threadCounter--;
  }
//
std::sort(recipeAndRating.begin(), recipeAndRating.end(),
std::pairint>& l,
int>& r) {
> r.second; });
//
std::cout << "\nLooping through ranked
for (auto it = recipeAndRating.begin(); it != recipeAndRating.end(); it++) {
std::cout << "Recipe=" << it->first
    << ", rating=" << it->second << std::endl;
}
```

```
}

};
```

The mutex is a locking primitive, and the easiest way is to create a lock_guard with the which will give you restrictive access to the variables. In our example here, we create a number of threads (one for each entry in our list of recipes), and in the thread, we generate a random rating and add that to our container (which is a just so we can sort it). That way, despite our container not being thread-safe, we will not have any problems with multiple threads updating at the same time.

Our random output shows that for each recipe, we had to think about the rating for different periods. Once we decided on a rating, we safely added it to our container, and then afterwards, we printed our ranked collection. By using the mutex, we did not have any locking issues, as shown in Figure

Looping through recipes
Thinking about rating on lasagna for 1 seconds
Thinking about rating on broiled chicken for 5 seconds
Thinking about rating on falafel for 2 seconds
Thinking about rating on ravioli for 2 seconds
Looking at ratings
Recipe=lasagna, rating=1
Recipe=falafel, rating=33
Recipe=ravioli, rating=47
Recipe=broiled chicken, rating=81
Looping through ranked ratings
Recipe=broiled chicken, rating=81
Recipe=ravioli, rating=47
Recipe=falafel, rating=33
Recipe=lasagna, rating=1

Figure 8.7: Output from lookAtMutexMap

The other class that is often used is called which was in the C++17 release. This synchronization primitive is not like the normal mutex that always just gives exclusive access. This has shared access and exclusive access, so you can have multiple reads looking at the data, but only one thread writing the data.

Our semaphores, on the other hand, are more used for notification versus having exclusivity to a resource. You create a counting_semaphore giving it the maximum number of threads that can access it at the same time. If you call it decreases the count, and then when a thread is done, it calls release() to increment the count. If enough threads try to acquire simultaneously, then once the count gets to 0, the thread will block until it is allowed access.

Simply stated, a binary_semaphore is an alias of counting_semaphore with a maximum of 1.

Our example code will create several threads, but the counting_semaphore will only allow access to 2 at a time:

```
{



{

counter{ 2 };

  threadNumber);

counter.acquire();

  threadNumber);

// use the resource for a brief while


counter.release();

}


{
```

```
<< << << __FUNCTION__ <<

=


for n = n < thread_count; n++) {

  threads[n] = n);

  }


}

};
```

You would be correct if you guessed that if we created six threads, but only allowed access to two at a time, that there would be some delays. Our output in Figure 8.8 shows:

```
Thread#10816. 12:00:09.557: Thread counter=2, Started thread
Thread#1776. 12:00:09.557: Thread counter=0, Started thread
Thread#7940. 12:00:09.557: Thread counter=1, Started thread
Thread#14188. 12:00:09.557: Thread counter=3, Started thread
Thread#9756. 12:00:09.557: Thread counter=4, Started thread
Thread#31940. 12:00:09.557: Thread counter=5, Started thread
Thread#10816. 12:00:09.558: Thread counter=2, acquired
Thread#1776. 12:00:09.559: Thread counter=0, acquired
Thread#7940. 12:00:10.566: Thread counter=1, acquired
Thread#14188. 12:00:10.566: Thread counter=3, acquired
Thread#9756. 12:00:11.571: Thread counter=4, acquired
Thread#31940. 12:00:11.571: Thread counter=5, acquired
```

Figure 8.8: Output from lookAtCountingSemaphore

If you look at the times when the threads acquired the resource, the first couple of threads got it right away, while the others had to wait.

Aside from the acquire function in our semaphore classes, we have some other options as well:

It has the same goal as acquire, but it does not block trying to get access to the resource. It returns a Boolean stating whether it was successful, and if the thread got the resource, it will decrement the counter and return true. If not successful, then it will just return false.

It is similar to in that it returns a Boolean if it successfully got the resource. This function also has another parameter stating the duration that it will block waiting for the resource (if it has to wait).

It is also similar. It too will return a Boolean denoting success. Instead of stating a relative time, it will wait and block, and for this function, you need to provide an absolute time that it will block for.

We also included an updated version of our outputText function. Since different threads were trying to output to std::cout at the same time, the result was a bit garbled. We included the header, and created an osyncstream for our output, and that made it look more organized. Synchronization is everywhere!

Recipe 8.6: std::atomic, and std::atomic

Problem: We want to be able to manipulate shared_ptr objects atomically.

Solution: The C++ 20 release has std::atomic<> for shared_ptr and

Before we discuss the new templated class, we will go over and The template std::atomic has been around since C++11. A quick description is that it allows atomic operations to be done on that type so that you do not have to worry about data races or coding extra locking.

simpleAtomicNumber;

{

{

simpleAtomicNumber +=

}

{

```
<< << << __FUNCTION__ <<

simpleAtomicNumber =

=



for n = n < thread_count; n++) {

  threads[n] =

  }

for n = n < thread_count; n++) {









  }

<< "Result="

  << << << simpleAtomicNumber;

}

};
```

In our sample, we have an that we create 100 threads, and each increment by 0.42. Naturally, our result is as shown in Figure

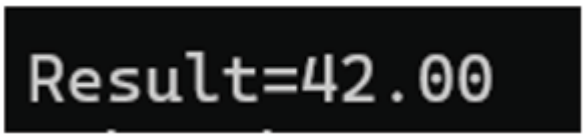

Figure 8.9: Output of lookAtSimpleAtomic

There were a number of member functions with atomic<> when it was initially introduced:

A void function that sets the atomic to the value provided. There is also an operator= which gets the same effect.

A function where it returns the current value.

Where you pass in a new value, and it returns the previous value it held.

If the templated type is numerical (integral or floating), then there are the usual among some others, but not

If you want to do bitwise operations, there is and

You can also have but there is no atomic increment.

You can have std::atomic but of course, it does not have those operations that integer types have. There is also the class std::atomic_flag that would seem similar to but atomic_flag does not have the load and store operations. Instead, it has the following:

Set the flag to false.

Returns the value of the flag.

Sets the flag to true, and returns the previous value.

The type atomic_flag also has the functions: and which we will be talking about shortly. Some people prefer because that is guaranteed to be lock-free.

We can almost get to our topic; but since we have talked about atomic, we can briefly talk about The templated type shared_ptr is a smart pointer that allows for shared ownership. It has an internal reference counter, allowing for several shared_ptr objects to own the same underlying object. It knows the last remaining owner of the object needs to clean it up once it is finished with it. Just remember that while shared_ptr can safely do reference counting, it does not guarantee thread safety for the object it is pointing to. A few of the functions for a shared_ptr are:

There is an operator= used for assignment.

Replaces the managed object with the parameter given.

Exchanges the stored pointer values with the parameter given.

Returns the stored pointer.

Returns the number of instances managing the current object.

The std::weak_ptr templated type has similar functionality to shared_ptr, but a weak_ptr does not manage the lifetime of an object. When the last reference of a weak_ptr goes out of scope, it does not affect the life of the underlying object.

In C++20, they made specializations of std::atomic for std::shared_ptr and This allows users to manipulate the objects shared_ptr and weak_ptr are pointing to atomically. Most typical atomic functions are available to the specializations, and actually (so far) there are no additional member functions added. Some of the details for these specializations are as follows:

You can construct an atomic with a but you cannot construct a new one by providing an existing atomic.

Similarly, you can use the assignment operator by providing a smart pointer but not by having the right-hand side be atomic.

The function store is given a smart pointer.

The function load returns a copy of the

The function exchange works as expected; using the provider smart pointer to replace the value, and returning the old value that it had.

Recipe 8.7: std::atomic::wait, and std::atomic::notify*

Problem: We want a mechanism to block until we see a value has changed, and then "tell" or unblock other thread or threads.

Solution: C++ 20 added functions to std::atomic such as and

While other recent atomic changes have been appreciated, probably the biggest praise has been for its new notification system. Now atomic types have a built-in way to tell others that are waiting for them when they are done.

An atomic can tell others that may be blocked waiting for a signal they can now proceed, with a notify function. The function notify_one tells a thread that is waiting, that it can now continue. If no other listeners are waiting, then nothing happens. If a thread wants to signal all threads that are waiting, it can call

The listeners that are waiting for a signal, would use the wait function. The parameter is the old value; since the listener may not know what a new value is, it supplies the old value and will be told when it is not that value. It is important to realize that although it does a bitwise comparison on the value, it does not do comparison functions, and certainly does not include logic in its determination on whether it can continue.

Often, the signal means that something has completed, and another thread can perform an operation, but that does not always have to be the case.

Sometimes it is to refill a queue, or to gather more data, and that is why the type and value you will be waiting on is an important aspect to ponder.

In our example (which is similar to we are cooking several recipes, and we want to know when we are done:

```
allRecipesCooked;

recipeCount;

recipesCompleted;

{

=




recipeCount++;

= recipeCount.load();

to cook for recipe recipeId);

// things take different times to cook

% LongestWait));
```

```
cooking recipe recipeId);

recipesCompleted++;

recipeCount--;

if == {

allRecipesCooked =

allRecipesCooked.notify_one();

}

}

{

=


for n = n < thread_count; n++) {


  threads[n] =
```

```
}


<< "We have cooked " << recipesCompleted.load()

 << " recipes, time to eat!" <<

}
```

We have a couple of atomic variables: One for the number of recipes we have completed (which will increase), and one for the active number of recipes being cooked (which will increase, then decrease when done). When our recipeCount returns back to 0, we set our atomic for and use that to notify whoever is waiting that we are done. In this case, we used because we knew there was only one thread waiting, but we could have used and we would have got the same result.

Atomics not only make things safer, but could allow for writing code faster. Not having to worry as much about code for locks or waiting for signals, it should help your productivity. Sadly, albeit rare, one thing to be aware of is a situation called false sharing. This is a scenario where adjacent memory in the cache gets modified and triggers a reload, hampering performance. Sometimes you may not be aware of it, but if you do have the issue, you can weigh the consequences compared to solutions such as padding your object size to be larger so there is no false triggering.

The main thread is waiting to be notified when allRecipesCooked is not false, and then we can output the number of recipes completed, as shown in Figure

Thread#31280. 23:07:19.002: Thread counter=3, starting to cook for recipe
Thread#16360. 23:07:19.003: Thread counter=4, starting to cook for recipe
Thread#28948. 23:07:19.003: Thread counter=8, starting to cook for recipe
Thread#25004. 23:07:19.002: Thread counter=1, starting to cook for recipe
Thread#2744. 23:07:19.003: Thread counter=9, starting to cook for recipe
Thread#27524. 23:07:19.002: Thread counter=2, starting to cook for recipe
Thread#17776. 23:07:19.003: Thread counter=5, starting to cook for recipe
Thread#22292. 23:07:19.003: Thread counter=6, starting to cook for recipe
Thread#32164. 23:07:19.003: Thread counter=7, starting to cook for recipe
Thread#13808. 23:07:19.003: Thread counter=10, starting to cook for recipe
Thread#31280. 23:07:19.144: Thread counter=3, done cooking recipe
Thread#28948. 23:07:19.203: Thread counter=8, done cooking recipe
Thread#25004. 23:07:19.282: Thread counter=1, done cooking recipe
Thread#2744. 23:07:19.330: Thread counter=9, done cooking recipe
Thread#17776. 23:07:19.536: Thread counter=5, done cooking recipe
Thread#32164. 23:07:19.598: Thread counter=7, done cooking recipe
Thread#16360. 23:07:19.630: Thread counter=4, done cooking recipe
Thread#13808. 23:07:19.646: Thread counter=10, done cooking recipe
Thread#22292. 23:07:19.710: Thread counter=6, done cooking recipe
Thread#27524. 23:07:19.797: Thread counter=2, done cooking recipe
We have cooked 10 recipes, time to eat!

Figure 8.10: Output from lookAtSimpleWait

Recipe 8.8: std::latch

Problem: We want a downward counter used to synchronize threads.

Solution: The C++ 20 release has

A bit opposite of the atomic::wait function, the latch is waiting for a value, although the value it is waiting for, is always zero. You initialize the variable with a number, and there is a count_down() function that decrements and a wait function that blocks until zero. This also has a that tests if the value is zero, and an arrive_and_wait function that does the double duty of decrementing and then blocking until zero. You could get the same effect with an and/or counting threads, but this is a nice simple mechanism that does not focus on how you were getting the work done, but whether or not you are done yet. It should be kept in mind that you cannot use a latch more than once, when it counts down to zero then it is done.

recipesToCookAndLatch[] = { "broiled "ravioli" };

recipesToCook{ };

{

food) {

=

```
  to cook recipe: " + food);


// things take different times to cook

% LongestWait));

  cooking recipe: " + food);

recipesToCook.count_down();

}


{

=


for n = n < thread_count; n++) {

  threads[n] = recipesToCookAndLatch[n]);

  }
```

```
recipesToCook.wait();

<< "We have cooked " << thread_count << " recipes, time to eat!" <<

}

};
```

In our example, we have a number of recipes to cook. We create a thread for each one, and they all take different times to cook, but once they are done, we decrease the Meanwhile, in our main thread, we wait for all the recipes to cook, at which time we can dine. Refer to Figure

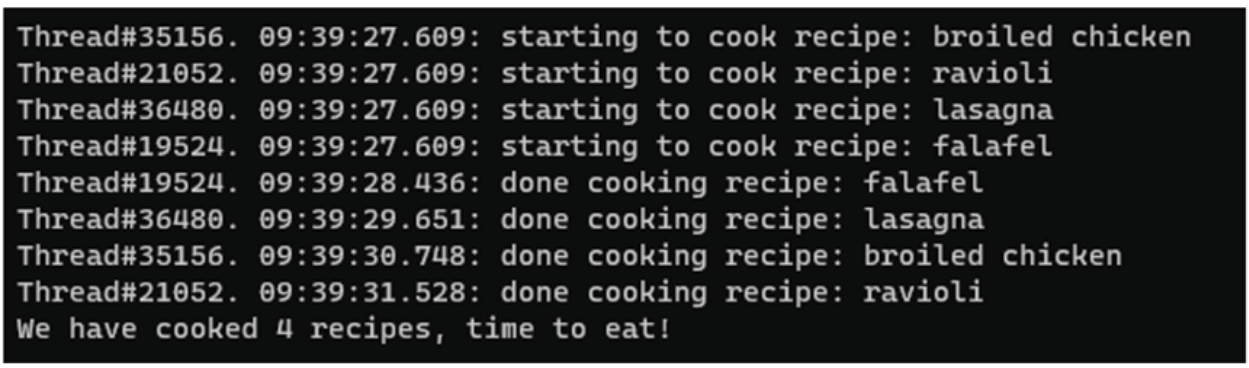
Thread#35156. 09:39:27.609: starting to cook recipe: broiled chicken
Thread#21052. 09:39:27.609: starting to cook recipe: ravioli
Thread#36480. 09:39:27.609: starting to cook recipe: lasagna
Thread#19524. 09:39:27.609: starting to cook recipe: falafel
Thread#19524. 09:39:28.436: done cooking recipe: falafel
Thread#36480. 09:39:29.651: done cooking recipe: lasagna
Thread#35156. 09:39:30.748: done cooking recipe: broiled chicken
Thread#21052. 09:39:31.528: done cooking recipe: ravioli
We have cooked 4 recipes, time to eat!

Figure 8.11: Output for lookAtSimpleLatch

Recipe 8.9: std::barrier

Problem: We want a mechanism that blocks a group of threads, until all threads have reached that barrier.

Solution: C++ 20 has a new templated class

The barrier class gives us a way to block a group of threads at a certain point, so none can proceed until they are all done. We need to initialize the barrier with a known count, and what callable to execute once they are all there. Unlike barrier is reusable for different stages. Both latch and barrier are focused on a counter, versus the actual management of the threads. Thus, you do have to be careful that there is no missing exception handling or logic flaws. Otherwise, the barrier may never be met.

Our sample here is a little bit different; we have a number of guests arrive from various distances. Once they all arrive, we can all eat. Some eat faster than others, so we will wait again before we move onto another stage (whether that is coffee, dessert, or everybody leaves).

```
guestsForDinner[] = { "Doug" };

noexcept {

status = "all
```

```
/* can change the value of the static,
  and when it is called again the status will have changed */
status = "all
}

barrierForGuests{ completionForAllGuests };
{

guest) {
=


  for arrival of guest: " + guest);
// people live at different distances
% LongestWait));
  of guest: " + guest);
```

```
barrierForGuests.arrive_and_wait();

// people eat at different speeds

% LongestWait));

  + " is eating

barrierForGuests.arrive_and_wait();

}



{

=



for n = n < thread_count; n++) {



  threads[n] = guestsForDinner[n]);

  }

for n = n < thread_count; n++) {
```

```
  }

<< "Hopefully all " << thread_count << " guests enjoyed their meal!" <<

}

};
```

We set up a barrier with a count, and start a thread for each guest. They all are passed to a callable, where each guest/thread will take a different duration to arrive, and a different duration to eat. For those durations, we use a barrier so that they all finish that stage at the same time, as shown in the following Figure

Thread#38788. 19:37:32.421: wait for arrival of guest: Bob
Thread#27844. 19:37:32.421: wait for arrival of guest: Doug
Thread#33524. 19:37:32.421: wait for arrival of guest: Charles
Thread#31676. 19:37:32.421: wait for arrival of guest: Allen
Thread#38788. 19:37:33.528: arrival of guest: Bob
Thread#31676. 19:37:34.964: arrival of guest: Allen
Thread#33524. 19:37:36.522: arrival of guest: Charles
Thread#27844. 19:37:37.087: arrival of guest: Doug
Thread#27844. 19:37:37.087: all arrived
Thread#33524. 19:37:37.349: Charles is eating dinner
Thread#27844. 19:37:37.381: Doug is eating dinner
Thread#38788. 19:37:37.396: Bob is eating dinner
Thread#31676. 19:37:38.159: Allen is eating dinner
Thread#31676. 19:37:38.160: all ate
Hopefully all 4 guests enjoyed their meal!

Figure 8.12: Output of lookAtSimpleBarrier

We could add in other stages in our waitForGuest function, and we probably would implement a cleaner implementation in that gets called when all threads meet at the barrier. This shows though it can be handy to have a simple mechanism to synchronize threads.

Recipe 8.10: std::atomic_ref

Problem: It would be handy, and performant, to have an atomic that worked easily with references.

Solution: C++ 20 included the templated type

We have seen how the atomic type is extremely helpful when dealing with a wide variety of types. We have seen it has specializations for Boolean, integers, smart pointers, and so on. What it does not handle well, is when you are working with references. Instead of creating more specializations for references, they created a new template called A few things to note about atomic_ref are as follows:

Unlike std::atomic_ref is copy constructible.

While any atomic_ref is holding an object, you should only interact with that object via

An object reference should be held by only one atomic_ref at a time.

Similar to there are specializations for (most) integers, and floating types. Having these specializations allows you do perform math operations on the references.

For a brief example, let us do something similar to an earlier sample:

```
{

0 };

};

simpleAtomicRef{ myValue.value };

{

{

simpleAtomicRef +=

}

{

=

for n = n < thread_count; n++) {
```

```
  threads[n] =

  }

for n = n < thread_count; n++) {



  }

<< "Result="

  << << << myValue.value;

}

};
```

Again, we have a bunch of threads that are calling a function that increments the value. The difference here is that the value being worked on is an which was initialized with a variable that lives in another structure. We get the same output, but we are incrementing an and not an atomic of a normal type.

Questions that are often asked are: which mechanism is better, and which mechanism should be used? The answer, naturally is that it depends on the

situation. All of the mechanisms are still used, depending on the scenario. It depends on what you are trying to lock, how busy is the system that you are trying to lock the object, and other questions like that. We can create a chart to review them, but there are nuances to each:

each:
each: each: each:

each: each: each: each: each: each: each: each: each: each: each: each: each: each: each:
each: each: each: each: each: each: each: each: each: each: each: each: each:

Table 8.1: Quick summary of some concurrent processing mechanisms.

Conclusion

We covered a lot of ground in this chapter. We dove into some of the new features in recent releases, and reviewed some of the concurrent basics along the way. Obviously, it is important in a different variety of applications to plan how you are going to tackle concurrency. Once again, we see there are different approaches to tackling a problem, and so, it is important to understand what nuances you might have, or need, to see which approach would work best for you. There are still pretty big concurrency features that are being talked about for upcoming releases, but the additions fill in some of the gaps and provide good building blocks for your solution.

Points to remember

The new std::jthread is a simple replacement that provides safety, and has extra features you can use.

The related and stop_callback allow you to gracefully exit out of a thread, without much effort.

The std::semaphore class is a good lightweight replacement for the heavy and slow So, if you can use that instead, your application will perform better.

There are a number of new atomic functions and mechanism that should make your code work much better with dealing with data in threads.

The latch and barrier classes allow you to set markers in your code where you expect something to happen, so it gives more flexibility to your application.

References

CPPreference. (n.d.).

GeeksForGeeks (n.d.). Retrieved from https://www.geeksforgeeks.org/mutex-vs-semaphore

Wikipedia. (n.d.). Retrieved from https://en.wikipedia.org/wiki/False_sharing

Join our book’s Discord space

Join the book’s Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 9

Coroutines

Introduction

Coroutines were another part of the Big 4 features of C++ 20. Some say it was the hardest feature to implement; some say it is the biggest shift in how code was written before, and some say that it is the best feature of that release. There certainly are a number of pieces that we will need to look into, and learn about. While in the last chapter, we talked about a bunch of new features that will help you create asynchronous code better, coroutines allow you to declare asynchronous operations. Coroutines are not always asynchronous, but they can be coded that way. In a way, the coroutines in your code look like blocking code, but without the blocking. Creating coroutines will give your code better responsiveness, and so the end users will appreciate that. The IT department that has to make sure your application can scale will also appreciate it as well. C++ 11 helped a lot by making asynchronous code cleaner by using lambda expressions, but the C++ 20 and 23 releases are designed to make code simpler and yet cleaner. Sometimes there can be an easier transition from non-asynchronous code to using coroutines, than switching to asynchronous code. Moreover, some love the coroutine flow because it is easy to continue from where you left off. While coroutines did not change much in C++23, there were some features that people wanted to have added but, in the end, they were not included.

Structure

In this chapter, we will cover the following topics:

Recipe 9.1: What is a coroutine

Recipe 9.2: Components of a coroutine: The promise

Recipe 9.3: Components of a coroutine: The state, and the handle

Recipe 9.4: Coroutine restrictions

Recipe 9.5: Generators

The good news is, except for generators (which none of the major compiler makers have made feature complete yet), the rest of the coroutine features seem complete.

```
Chapter9_3_1::RunSimpleGeneratorWithoutKnowingLimits();
```

```
Chapter9_3_2::RunIntGeneratorWithIterator();
```

```
Chapter9_3_2b::RunMenuGenerator();

Chapter9_3_3::RunGenerator();

Chapter9_4::CallRenamedGen();

Chapter9_5_1::FillWithGenerate();

Chapter9_5_2::FillWithGenerateUsingGenerator();

Chapter9_5_3::RunNumberGenerator();

}
```

Objectives

Coroutines are a big feature, but they have a large learning curve. It is a different paradigm, where it is not asynchronous code, but allows you to structure your code to be more effective. By the end of this chapter, the reader will have an understanding of what coroutines are, and the power they can provide.

Recipe 9.1: What is a coroutine

Problem: We want a flexible mechanism that allows us to call code multiple times, and it will just pick up where it left off.

Solution: C++ 20 has created coroutines.

Coroutines have actually been around for a long time. The term was first coined in 1958, and used in one of the early programming languages Simula Since then, Coroutines, in one form or another, have been in languages such as Go, Kotlin, Ruby, PHP among many others. In that respect, C++ is rather late to have such a feature. Those that know C# would relate it to the yield statement. To give a brief description of what they are, we can look at the following Figure

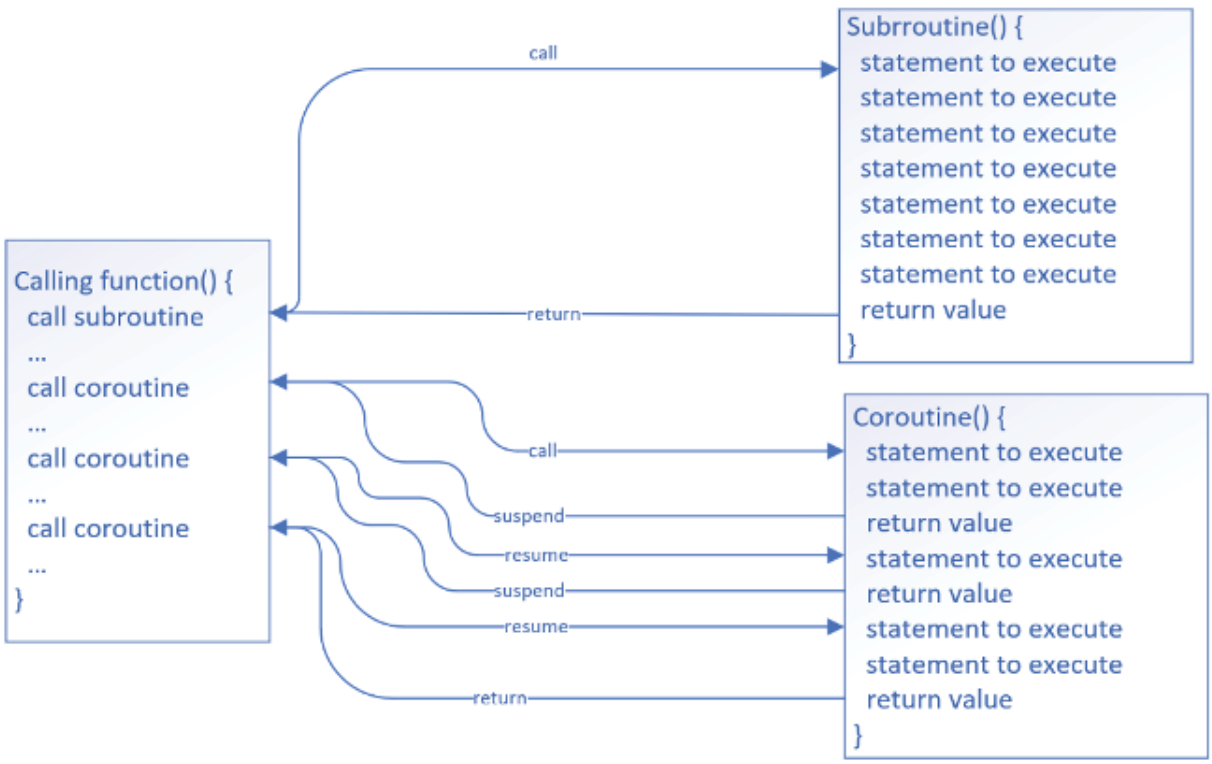

Figure 9.1: Comparing a subroutine to a coroutine

In a subroutine, we can see our calling code execute all the statements in the subroutine, before returning a value. In C++, we call the subroutines functions. For a coroutine, the calling function calls the subroutine, and it executes some code, and then it says it is done for now. Execution continues with the caller. The caller can call the coroutine again, and it continues where it left off. The caller can continue to call the coroutine until there is no code left to execute, where execution just continues back with the calling function.

One downside of the idea of coroutines, is that you sometimes need to do some coding on the calling side, as well as the coroutine side, to get it to work effectively. Other times though, especially in cases where the coroutine has an iterator, that the caller seems like it is in a normal loop, and the code is very clean. There are many ways for the coroutine to manage what they produce, and most times the caller will not know how the coroutine is implemented, or even if it is one.

If we think of a food analogy, imagine a breakfast restaurant assuming how many orders for eggs it will sell that day, cooking all the eggs up all at once, and then waiting for the customers to arrive and order them. Instead of doing all the work up front, the coroutine restaurant might cook a few eggs up when the restaurant opens, and will keep those warm waiting for orders, but otherwise it stops cooking and waits for somebody to request them to cook more eggs. The end result is the customers will be happier because they received a better product, and the restaurant owner will be happy because they did not waste a lot of eggs that were cooked ahead of time but never served. Coroutines allow you to provide some data back to the caller sooner, but not always to keep looking for data to serve back. This lets other code execute, and help the program flow better.

The coroutine does not execute in a different thread. As normal, the calling code sits there waiting for a response from the coroutine, but when the coroutine returns a value, its execution is suspended, until it is asked to continue (when it is called again). Normally, when C++ calls a subroutine (aka function), it also remembers where the caller is by means of saving information on the stack. When debugging C++ code, you will recall that we can see the history of who called what, by looking at the stack-trace. With coroutines, something similar happens, but the status, values of local variables, and other information is stored on the heap. The C++ implementation of coroutines is said to be stack-less. When a coroutine returns back to the calling function, it is said that the coroutine is suspended. Once it is called again, the coroutine is resumed. They call this cooperative multitasking.

While you can certainly code solutions that do not use coroutines, but get the same result, there are certainly advantages of using a coroutine, such as:

There is a defined mechanism for suspending and resuming within the same thread. You can do something that feels like concurrency, but since it runs in the same thread, you do not have to worry about data locks or data races.

Although there is a small number of pieces you have to define, the pattern would be easy to apply to different scenarios.

If you look at your calling code, you cannot tell that you are actually calling a coroutine. Hiding the implementation detail makes the code look very clean. You cannot tell if it is a coroutine by looking at its signature. You can only tell by looking at the implementation.

We do not have to worry about threads that are pre-emptive, and do not get the kernel involved (which could slow execution) (See references).

The idea of coroutines is to generate data and you do not have to allocate a lot of memory for a full result set. The coroutine can provide you with a single result for each execution. That means that you do not need to allocate a lot of storage if you can just process each individual result as you get it. Moreover, if you traditionally go get a bunch of data in a collection, then you probably had as much code managing the collection, as getting the data. Coroutines (in many cases) eliminate managing the collection completely.

The idea of using a coroutine can be very handy in several situations, such as:

When you have a function pointer as a callback. Sometimes you could be passing a callback to something you do not trust.

They can be used in event loops.

When you are parsing some data.

When you are dealing with a long, or even infinite, list.

When you are iterating over something.

Lazily computed sequences, also known as generators, are a big aspect of coroutines. We will be talking more about generators later.

Pipes.

Possibly state machines.

Other situations where you would want to temporarily suspend what you are doing.

A knock against C++ coroutines, is that there is a fair amount of boiler-plate code that needs to exist for it to work. We will talk about those different pieces in detail, but we can give a description for some of them here:

A handle for the coroutine: The calling code will need a mechanism of calling the coroutine, and when to call it again (or to resume it, as it is typically described). This handle is non-owning, and so it will need to be properly destroyed when the caller is done with the coroutine.

A promise object: The user defines what is in this, but there are rules to follow. It is used within the coroutine and is used to return the result. It has no relationship with

The coroutine state: As mentioned, this normally lives on the heap, and contains the promise object. If there is a change, the compile will optimize it and it will end up using the stack. However, in terms of the application code, that does not matter. It also contains the coroutine's local variables, and other arguments given to the coroutine.

You may have asked yourself "if the calling code does not know if it is a coroutine, how can you (or the compiler) tell?" It is from one of the 3 new keywords added to C++20: and If you code uses one of more of those co_

keywords (and follows all of the other rules), then the compiler will know it is a coroutine. Here are the stages:

The keyword co_yield is used when you want to return from a coroutine, but you are not necessarily done providing all the data (or at least you do not know yet if you are).

The keyword co_return signifies that you are done returning data. You can use it with a final value. For example, co_return 42; or a simple co_return; to say there is no more data to provide. If your generator function has performed a and then the function ends, there is an implicit co_return added.

Lastly, the co_await operator suspends execution of the coroutine, until it is resumed. The co_await can have a parameter that will return an awaitable (which we will talk about soon). There are predefined ones, or you can create your own, depending on your needs. While co_yield will return a value, and co_return tells the caller "I have no more values to return", co_await allows the coroutine to tells the caller "I currently have no more values to return (or want to take a rest from returning values) but you can call me again later and I will resume to look for more".

One last point to mention about coroutines is the difference between stackless and stackful. Naturally a stack exists when a program is executed, but stackless means the couroutine does not rely on a stack while it is suspended. C++ coroutines are stackless, as they use the thread stack from the operating system that bundles up the variables within the method. Conversely, stackful is the type normal threading code would have. Stackful normally means more memory is allocated to handful the entire call stack, while stackless need less memory to manage just the coroutine frame. A stackless frame can be thought of as an object. It has values for the method’s variables for that current point in time.

We have talked enough in an overview about coroutines. Let us start talking about the individual pieces.

Recipe 9.2: Components of a coroutine: The promise

Problem: We want to learn about the parts that make up a coroutine.

Solution: A coroutine is mainly comprised of a promise, a handle, and the state. We will first learn about the promise and the state type that holds it.

The initial challenge with building (or learning how to make) a coroutine, is that there are a few different pieces, and so it is hard to dive in and make something simple on which to build off of. The pieces are not complicated, just something that takes a bit of time to learn.

There is no defined sequence in which to talk about the components, so we can talk about the promise object first. The promise (which is not associated with is defined, but used inside the coroutine state object, by the coroutine handle. Some even call it the promise type, mostly because by definition, the code requires the struct named The is very different than most other C++ paradigms. There is a fair number of rules that your code has to follow for it to compile properly, which adds to the learning curve. You can use a different name for it, if you use the template but for now, we will keep it simple. You never interact with the as it is there for the coroutine state to use. If your coroutine returns data, or if you need to give data to it, then those variables will be defined in the promise. When a coroutine returns data, then they often called generators. If a coroutine does not return data, they are called tasks. The logic or flow does not change though, whether you are building a task or a generator:

Before we get into the requirements of the let us briefly mention about the return object. You define whether it is a class or struct, and you can name it whatever you want, (although in lots of sample code, you will often see it called something plain like

There are several methods the promise_type struct requires, as shown in the following snippet:

```
{



{ }

noexcept { }

{ }

// if a task; not returning a value

// void return_void();

// if a generator, returning a value



};
```

The function as it indicates, provides the return object which is what contains the It also will contain the generated value the caller wants.

The functions initial_suspend() and final_suspend() have a similar return type. The initial_suspend is called when a coroutine is at the beginning of execution. It is called after the promise_type object is constructed, and after a call to It will then call initial_suspend and co_await the response. This will tell the compiler whether the coroutine can just jump into the execution right away, or whether it needs to be told to start/resume. Conversely, the final_suspend() states what it should do during the processing of the This could be for any cleanup required in your return object/promise. It is also called if there is an exception, after calling it will call

This brings us to the next function; also, as you would think, if there was a problem during the execution, the coroutine framework calls You could rethrow, or save up the exception for later, or the worst case would be to terminate the application. Something to be careful of, is that you do not want to rethrow, or throw a new exception if there is a chance the caller has defined noexcept on the coroutine call. Some people code that the ReturnObject has in which case, you could return an object that held a normal integer, or that returned object could hold an exception, and the caller would check on the call's success.

The functions return_void and return_value should be pretty self-explanatory. You just have to include one, but not both of the functions.

Let us go back to initial_suspend() and to look at another aspect of coroutines. You can see in our promise_type definition previously, that we return std::suspend_always and These are a couple of pre-defined awaitables from the header, that, similar so have a number of functions that must be defined. You can define them in a couple of ways, and the compile will figure it out. You can use auto as the return type and be explicit with the return, or explicitly state the return type, and use an empty return statement. Either works, but it is probably cleaner to be consistent (unlike below, just done for demonstration purposes):

```
noexcept { return {}; }

noexcept { }
```

We do not need to get into the awaitable functions in detail, but a definition would be:

```
{

{ }

*handle;

h) { *handle = h; }

{}

};
```

The function of focus is the which indicates whether to wait (or await to be precise). The difference between the two predefined types, is that std::suspend_always means it always wants to wait, (or is ready to wait), and so await_ready returns false. For that is always wanting, or ready to wait, and so in that case await_ready() returns true.

Going back to our if we use initial_suspend() to return that would mean once the coroutine is initialized, initial_suspend says do not start running the coroutine. We want to resume it before continuing (even though we have not done much yet). In our final_suspend() function, we are returning which says we do not have to wait before continuing. Once we perform co_return we do not have to wait to resume again.

These are not base classes or base structs. You can define your awaitable type, your type, and your return object. There are just rules that you have to follow about the functions you need to define inside them. On a side note, if you think: If there is no base types, how does the compiler know what to expect?

Without digging deeply into the answer, defining an entity with certain expectations of what functions exist, it sounds like what we were talking about in the first few chapters about concepts we have in this book. Another indicator on how powerful a feature that is.

One advanced topic about promises to mention, relates to transformations. The promise can also have a function named If that exists, then every time a co_await occurs, it calls the await_transform function, and the object it

returns is what is awaited on. This allows you do add some extra functionality when the coroutine is suspended.

Recipe 9.3: Components of a coroutine: The state, and the handle

Problem: We want to learn about the parts that make up a coroutine.

Solution: A coroutine is mainly comprised of a promise, a handle, and the state. We will now learn about the state and the handle.

Now that we have the promise, let us look into the part that holds it, the coroutine state. We define the basic functionality, but of course when it is compiled, it builds in a lot of extra functionality. Our state type will contain our handle, as well as the value that we generate which we are returning to the caller. There are quite a few headers to include, and a lot of which we will be using is in the new header. Let us define what headers we will be using in this chapter:

```
#include

#include

#include

#include

#include

#include
```

```
#include

#include "cassert"

#include

#include

#include
```

Here is our first full coroutine, with a couple of ideas that was suggested in a C++ talk (See references):

```
T>

{

{

noexcept {

return };

    }

{ return {}; }
```

```
{ return {}; }

noexcept {

        }

T& value) noexcept {

            result = value;

return {};

        }

        T& {

return result;

        };

{}

        T result;
```

```
    };

    ~Generator9_3_1() {

if coroHandle.destroy();

    }

const {


if ==

{

coroHandle.resume();


        }

    }


promise) noexcept :

coroHandle{ }
```

```
    {

    }

coroHandle;

};

{


static {

for = i < i++) {


        }

    }


{

<< << << __FUNCTION__ <<


gen = MakeIntGenerator();
```

```
for x = x < x++) {

<< gen() << '

        }

<<

    }

};
```

We have created a templated type, although it was not essential. We have our promise_type as required, because we did not use coroutine_traits to specify what the type would be called. Inside the promise, we have the functions that we have discussed. These dictate how the coroutine acts. We have a different syntax on being explicit on the return type (instead of auto) and an empty return just to show you a different style. Our struct has a (private) T result variable, which we set in our yield_value function and return in our getValue function. The other quirky syntax to note is in which creates a generator (which in this case, we named but you can name it whatever you want), and we pass it the The Generator9_3_1 constructor is a private constructor that takes the promise and creates a handle to it. At the end of the generator type, we have our variable for the Again, you can name it whatever you want; many call it handle or

Another item to point out about is that we are passing in a const reference, but that means we are copying the value into the local variable. Sometimes this seems like a wasteful copy, but it is far safer, and a lot easier when dealing with simple value types. The value that is passed in co_yield is still accessible during suspension, while inside the coroutine state. it could still be looking at the value, until the promise is complete. You just want to be very careful if trying to save a copy instruction and passing by reference.

With this implementation, the handle is private and not seen by the caller. Thus, the result is cleaner code and a nice example to reuse elsewhere (which you will be doing a lot). Some examples create the handle and pass it into the generator, which might be a requirement in some places, but hopefully not a necessity in many. Another nice part is the implementation of the operator() in the generator. You can see that it calls resume on the handle, allowing it to continue execution, and calls

In our MakeIntGenerator() function, it returns the generator type and performs a So aside from the you would not know it is using a coroutine. Perhaps some will say it looks a bit too magical, but you will see in the RunSimpleGeneratorWithoutKnowingLimits() function, that we create the operator and call the functor (the operator() overload). Calling the functor on the generator gives us the yield value that we print out. This code works, and looks simple enough, but it has problems. Note in the that we loop from 1 through 4 (less than 5), and when we call the functor on the generator, we are in a loop from 1 through 2 (less than 3). Refer to Figure

Figure 9.2: Output from our initial coding of RunGenerator

Right now, if we call the functor more times than the number of values it yields, the program will crash. What is even worse is that it will crash in a spot that is not in the promise where we are looking for an The caller does not know how many iterators it can do, and it will crash if it calls it too often. You could use this approach if both sides knew ahead of time the number of iterations it can do, or else if it was an infinite number of iterations, then it would not matter. In terms of what we are doing here, it is a nice start and something we can improve on.

It is a common programming paradigm about how to implement a solution for this. You could implement a function such as hasValue() that will say whether there is a value we can use, but a cleaner approach would be to follow what our other iterator types do, as shown:

```
while it = col.begin(); it != col.end(); it++) {

= *it;

    }
```

For reference, the coroutine_handle does have a boolean done() method, which can be used to see whether we can resume or not, in order to get another value. We can just add a function to our generator to check for that:

```
const {
```

```
coroHandle.done();

    }
```

Thus, that can be used in the calling code so that it would not throw an exception, as shown:

```
gen = MakeMenuGenerator();



for = i < 15 && gen.canResume(); i++) {

<< gen() <<

        }
```

For generators that we do not know the limit for (but a limit exists), adding this mechanism can be handy, or even at times critical. If we look at this code, even though we do not have 15 values, it will continue until it runs out. This works fine, but the code does not look as clean as it could be. Moreover, if you try to get a value without checking if you are at the end of the iteration, then it could throw an exception.

If we could check when we are done, we can just get the value when we need it. If we create another version of our generator, as follows:

```
T>
```

```
{

{

noexcept {

return };

        }

{ return {}; }

{ return {}; }

{



        }


T& value) {

            result = value;

isInitialized =
```

```
return {};

        }

// precondition for getValue: either have value or exception

        T& noexcept {

return result;

        }

{}

{

        }


        T result;

=

    };

{
```

```
Iterator() noexcept =

coroGiven) noexcept :

coro{ &coroGiven }

        {

        }

Iterator&, const Iterator&) noexcept =

        Iterator& {


assert(coro !=



coro->resume();

if {

=

            }
```

```
return

        }

{

assert(coro !=

        }

coro =

    };

    ~Generator9_3_2() {

if coroHandle.destroy();

    }

    Iterator const {
```

```
if

auto it = Iterator{ coroHandle };

{

        ++it;

}

return it;

    }

   Iterator {


return {};

    }

const {

if == {

coroHandle.resume();
```

```
        }
    }
const {
coroHandle.done();
    }

promise) noexcept :
coroHandle{ }
    {
    }
coroHandle;
};
```

The previous methods have not changed much. We added an isInitialized flag, which is set when we yield a value. The big change is that we added a struct called which is built with our coroutine handle. We have similar

usage of the handle within our plus check if it has been initialized. This means we can call the functor on our iterator, instead of our generator.

```
{


static {

for = i < i++) {


}

   }


{

<< << << __FUNCTION__ <<

= MakeIntGenerator();

for = i < 5 && anotherGen.canResume(); i++) {

<< anotherGen() <<
```

```
        }

<< << "Another pass, with iterator" <<

gen = MakeIntGenerator();

for it = gen.begin(); it != gen.end(); ++it) {

<< *it <<

        }

<<

    }

};
```

Now that our code looks better, let us try something else. We did not have to make a new MakeIntGenerator function, but we have not talked about food for a while. In the code sample below, we made just to show that a generator can yield values from different sources. When we iterate through we are returning a copy of the string, but in this simple example that seems fine. Most times, we would be performing a more complex operation, and probably returning something more detailed.

```
{


static {

menuItemsFromDatabase[] = { "Shrimp "Fresh Salad" };

itemRetrievedFromYesterdaysMenu =

co_yield itemRetrievedFromYesterdaysMenu + " sandwich, and

for : menuItemsFromDatabase) {


        }

    }


{

<< << << __FUNCTION__ <<

<< "Today's menu:" <<

gen = MakeMenuGenerator();
```

```
try {

auto it2 = gen.begin();



auto it = gen.begin();

for (; it != gen.end(); ++it) {

<< *it <<

            }

<< "\n" << "it is " << (it == gen.end() ? "Done" : "not <<

<< "it2 is " << (it2 == gen.end() ? "Done" : "not <<

for it = gen.begin(); it != gen.end(); ++it) {

<< "second pass" << *it << '

            }

        }

catch ex) {

<< "caught " << << ": " << ex.what() <<
```

```
        }

<< << "Today's menu again:" <<

gen2 = MakeMenuGenerator();

try {

for : gen2) {

<< item <<

            }

        }

catch ex) {

<< "caught " << << ": " << ex.what() <<

        }


<< << "Old way of looking at the menu:" <<

gen3 = MakeMenuGenerator();
```

```
for = i < 15 && gen3.canResume(); i++) {

<< gen3() <<

        }

    }

};
```

In our RunMenuGenerator() function above, you can see that we call begin() and end() as we would in many iterators, and we call the dereference operator to get the value. The variables were set for begin() and end() outside of the loop just to illustrate something. After we went through the loop once, we tried to go through the same generator again, but the output would have "second pass" to show us when that happened.

In our output in the following Figure we do not see any of that second pass output, nor do we see that we caught any exception trying. That means the good news is that, unlike other possible implementations, we did not throw an exception trying to call begin() more than once. The bad news is that a generator can only be used once, which is not a terrible, or a rare thing to occur. We can certainly create the generator again and run it, which we did to show the output in a for-each loop. That code seems perfectly natural, without knowing it came from a coroutine. We did a third loop, using our previous counter, to show that approach still works, but we still have the problem of not knowing how many items are going to be generated. The loop works, but not as well as we would want.

Refer to the following Figure

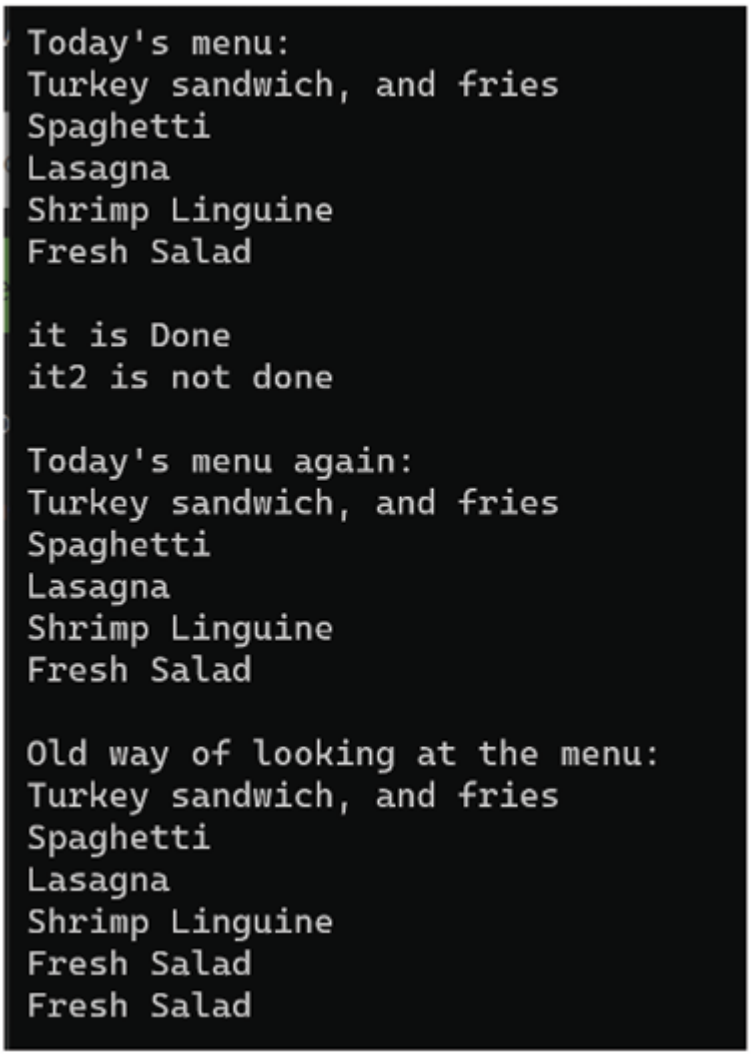

Figure 9.3: Output from our RunGenerator_v2

While we are here and can iterate through our entire generated list without throwing an exception. We can go back and look at something we have not talked much about yet. If we tweak our current code a little bit, as shown in the following snippet, we can see what happens if we change the result from suspend_always to

```
T>

{

{

noexcept {

return };

        }

{ return {}; }

{ return {}; }

T & value) noexcept {

            result = value;

isInitialized =

return {};

        }

noexcept {
```

```
        }

        T& noexcept {

return result;

        }

{}

{

        }

        T result;

=

    };

{
```

```
Iterator() noexcept =

coroGiven) noexcept :

coro{ &coroGiven }

        {

        }

Iterator&, const Iterator&) noexcept =

        Iterator& {

assert(coro !=



coro->resume();

if {

=

            }

return
```

```
        }

{

assert(coro !=

        }

coro =

    };

    ~Generator9_3_3() {

if coroHandle.destroy();

    }

    Iterator const {

if
```

```
auto it = Iterator{ coroHandle };

{

        ++it;

    }

return it;

  }

  Iterator {

return {};

  }

const {

if == {

coroHandle.resume();


    }

}
```

```
const {

==

    }


promise) noexcept :


coroHandle{ }

    {

    }

coroHandle;

};
```

Moreover, we do not have to show the output, mainly because there is none. If we say when we want to yield a value, that it does not have to pause before getting the next one, then it could iterate through the entire generated list before we actually use the result that we stored. There may be some circumstances that we would want something like this:

If we were going to store the generated values internally to feed into something else, then it may not matter if we have done the initial list.

Theoretically, if it is generating an infinite list, then we might want to see where we currently are.

However, these circumstances may be rare, and your coroutine state could look rather different. The final_suspend() function of the promise_type in most cases will return Moreover, if you rush the you can also be in a situation where the generator gets destroyed before you can use the value. That is when you could be doing some cleanup, and there is little repercussion on delaying it. The initial_suspend() might return suspend_never in some circumstances, depending on what you need to do. Again, there is the option for more complex situations to create your awaitables.

You can envision that these parts normally work in pairs:

The coroutine_handle and the promise are linked together. You get to the promise from the handle with the .promise() function, and from the promise, you get the handle via the aptly named from_promise() function.

Your coroutine gets a lot of its power from the awaitables.

The caller’s focus is getting the value it wants, out of the return type.

Your return type will often contain the handle, and the return type will also contain the promise.

Recipe 9.4: Coroutine restrictions

Problem: When trying to create a coroutine, we need to know what we cannot do.

Solution: A coroutine is powerful, but true; there are rules and restrictions that we will discuss.

We have already talked about some issues you could have with coroutines, such as if you used the wrong awaitable, or did not check if a value was usable, and so on. Those are more focused on how you design your code. There are a number of coding rules on what you need to do (or what you cannot do).

The first restriction is pretty easy; your main function cannot be a coroutine. This does not seem to be any hardship and so we will move on.

Your coroutine cannot have a (plain) return statement. With coroutines, it is or You might forget about the return now and then, but thankfully your compiler will remind you about it. It also cannot return a placeholder return type, such as or A trailing return type is allowed.

Another restriction is that you cannot use variadic arguments, but at least there is an easy way around that, using the template parameter pack, as shown:

// cannot have unnamed variadic arguments

```
// {

// but can have named template parameter pack

{

<< "Inside counter, going to start" <<

for = i < i++) {



<< "Inside counter, after await, now at: " << i <<

    }

}
```

Your coroutine cannot have constructors or destructors. Your task/generator type may need a destructor to destroy the along with other allocated variables.

The coroutine cannot be constexpr or

We have mentioned before about the promise requiring the name unless you use or some other mechanism to get around the name. Even a using statement will suffice, such as:

{

:

{

=

};

{

MyCoroutine { return { }; }

noexcept { return {}; }

noexcept { return {}; }

{}

```
{}

    };

};

{



{

<< << << __FUNCTION__ <<

        gen = i) -> Generator9_4::MyCoroutine {

<< i;



gen.resume();

gen.destroy();

    }

};
```

It was mentioned before, but you cannot have return_value and return_void in the same promise type.

Lastly, it probably is not something that you will normally trip over, but when the coroutine is getting destroyed, it only cleans up the local variables that were initialized before the last suspension point.

To end on a positive note, here is a list of where and how coroutines are allowed:

Coroutines can be freestanding.

Coroutines can also be static.

They can be a member function in a class, whether static or not.

Coroutines can be virtual.

Coroutines can even be a lambda expression.

You may nest coroutines, so in one generator you can co_yield values that you get from another generator.

Note: If you are interested in digging more into the potential of coroutines, you can look at a library called CppCoro created by Lewis Baker, one of

the people that created C++ coroutines. It can help explain a bit more, and you can leverage some of the features that might be coming in C++26.

Recipe 9.5: Generators

Problem: We want more flexibility with a mechanism to generate values.

Solution: The C++23 generator is a lot more flexible than the C++ version.

It is not fun when you start on a tangent, but before we talk about the C++23 feature of we should touch upon the C++17 feature of We can summarize std::generate() by saying that it is a way to fill a collection of known sizes by calling a function to give us a value to fill it with. A simple example would be:

{

{

=

}

{

```
<< << << __FUNCTION__ <<


col.end(), generateToFill);

for it = col.begin(); it != col.end(); ++it) {

<< "Number=" << *it <<

      }

   }


};
```

As you can see, we define a collection of a set size, and then call std::generate to say you can fill the values of that collection, based on the collection limits provided, with the generator function you provide.

Actually, third argument can also be a function object, so we can enhance our code by creating a class that has a function call operator:

```
{


   {
```

```
=

    }

{

initialValue = : value{ initialValue } {}

() {

return value++;

}

int value;

    };

    {

<< << << __FUNCTION__ <<
```

```
col.end(), generateToFill);

for it = col.begin(); it != col.end(); ++it) {

<< "Number=" << *it <<

        }

//


col2.end(),

for it = col2.begin(); it != col2.end(); ++it) {

<< "Number=" << *it <<

        }

    }

};
```

The execution results in the collections being filled, as shown:

```
Number=1
Number=2
Number=3
Number=4
Number=5
Number=6
Number=7
Number=8
Number=9
Number=10
```

Figure 9.4: Output from fillWithGenerate

Looking at the code, it seems like std::generate is a nice, but not groundbreaking feature. At least, we can show you how it works. Looking at the current status of what compilers have completed the C++23 features (See references), we have the following figure:

C++23 feature	Paper(s)	GCC libstdc++	Clang libc++	MSVC STL	Apple Clang	SunOracle C++ Standard Library	Embarcadero C++ Builder Standard Library
std::generator: synchronous coroutine generator for ranges	P2502R2 P2787R0						

Figure 9.5: Status of C++23 generator feature

This shows that although the feature is defined in the C++23 release, sadly we do not yet have any compilers that have implemented those features.

Refer to the following code:

```
#include

#include

{



#if >= 202207L

first)

    {

first++);

    }

#endif



    {
```

```
<< << << __FUNCTION__ <<


#if >= 202207L

for : |

<< num << '

<<

#else

<< "Feature not yet implemented." <<

#endif

    }

};
```

Without all of our #if checks, the code would not compile. Unfortunately, we do not get any output to show off this interesting new feature.

The generator class is based on ranges (which we will talk about in an upcoming chapter). It has a begin() and and hence, easily usable in a loop. It also has an empty() function to check if it has generated all values. Similar to

what we have with coroutines, but for simple cases, there is way less plumbing to create.

Conclusion

As mentioned, you need to brace yourself to learn about coroutines because it is an advanced topic which can be a bit confusing with all the various pieces. On the bright side, they do work well, and you will probably be reusing a lot of the code that you create, which is a nice way to enjoy the power of this C++20 feature.

There were some coroutine-related features that did not make it into C++23 (executors, or senders/receivers). Perhaps they will get into C++26, or maybe after that. Coroutines might even look a bit different with a few more releases down the road, but for today, they are still a powerful tool you can use.

Even though most people will not enjoy all the plumbing code required, some great benefits of using coroutines is that you do not have to worry about locking mechanisms, or how to set it up to avoid race conditions.

Points to remember

Coroutines were a C++20 feature, not connected to the C++17 std::generate feature, and only partially related to the C++23 std::generator feature.

The main components are the promise, the state, and the handle.

It also involves awaits, but often, you might use the included and

You can control the type names, but there are functions that must exist for your coroutine to compile. When it does compile, a bunch of extra code is added in.

There is a fair amount of plumbing you need to do, but often, you should be able to reuse that between projects.

The caller probably does not know it is calling a coroutine.

References

Wikipedia. (n.d) Origin.

YouTube. (n.d)

GitHub. (n.d.)

CPPReference. (n.d.)

Github. (n.d.). Retrieved from https://github.com/lewissbaker/cppcoro

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 10

Organizing Your Code with Modules

Introduction

Although still code-related, in this chapter we will look at a different paradigm and not focus on how to write some new code, but how to organize your code. On the surface, modules may not seem like a big language change, but they will certainly help you create better software. It is a way of defining a section of code, that does not rely on namespaces, but rather, helps you maintain your code better, as well as when you reuse other people's code.

Structure

In this chapter, we will cover the following topics:

Recipe 10.1: What is a module

Recipe 10.2: What the compiler does with a module

Recipe 10.3: How to use a module.

Recipe 10.4: Visibility and reachability

Recipe 10.5: Layout of a module

Recipe 10.6: Cyclical modules

Recipe 10.7: Importing the standard modules

Recipe 10.8: What are module partitions

At the time of writing this book the Microsoft compiler is mostly feature-complete, and clang and gcc are partially complete. The issue is, that we normally use the Compiler Explorer on https://godbolt.org to verify the status of the features of compiler, but they have not yet fully implemented the ability to use modules. It is best to check on

https://en.cppreference.com/w/cpp/compiler_support to see if your compiler has implemented modules.

{

}

Objectives

Modules are a relatively easy topic to understand and certainly welcome. Although there are many modern features in the C++ language, having to separate the code into header and implementation files seems rather odd compared to many other languages being used today. By the end of this chapter, the reader will understand more about what modules are, and how to create and consume them.

Recipe 10.1: What is a module

Problem: We want a better way to organize our code.

Solution: One of C++20's Big Four features is called modules.

Modules have been around in other languages for a while, so, C++ was a bit behind in that regard. The way C++ built things has not changed much since it was initially released decades ago. It was based on the C build system, which had the simple principle of declarations in a header file and the implementation in another file. Compilers got very good at optimizing the code, but aside from that, it was just like the basic steps of compiling the .CPP files, and then linking the object files into some form of executable. Many newer languages did not have that thinking of separating the declarations from the implementation, partially because other languages were newer and had the benefit of more computing power. Using header files for declarations does have some advantages, such as:

The process was fairly simple, which meant that it was easy to understand, and compilers could even process multiple header files at once.

Aside from inline code, it had a good separation from the implementation.

It was easily understandable for other C++ developers to know what was available for them to consume.

Unfortunately, the list of disadvantages of using headers was longer and more serious:

Including a header means a copy and paste of the header contents into your implementation file. The biggest complaint about all of this was that compilation times could get extremely long. If you have a header that is included in 50 source files, it means that the contents of the header are being compiled that many times.

Precompiled headers helped with the speed issue to an extent, but it could also get confusing, and sometimes relied on the sequence of header includes, which was not always consistent.

Even with pre-compilation, you can have many compiler errors due to the order in which you include your headers.

The blessing of separation can also be a curse. The files could get out of sync, or other problems like cyclical errors.

Most other modern languages had processors, or other tools, and comment-readers, that made the separation far less important.

Having C++ headers is good for other C++ consumers, but in this modern age of enterprise applications using multiple languages, there is no easy (or standard) way for other programming languages to understand the contents of the header files.

A huge issue is something called the One Definition Rule and with macros, it is too easy for different C++ types to have compiler issues, or worse, unforeseen runtime errors.

Headers also have a lack of encapsulation, which was one of the main objectives of the language. You are exposing some implementation to anyone who consumes your public header.

Despite them being around for a very long time, people wanted the system to improve, and finally, in C++20, they released the feature called modules. Modules are a way of solving the problems listed previously. There was a talk (see references) that said that while modules may not have been the C++20 feature with the most fanfare, not only are modules the most important feature of that release, but they are the most important feature since C++ was standardized. The argument behind that statement relates to modules:

They dramatically change the build system.

It will change how C++ code is written.

It will change how C++ code is parsed.

It can change how C++ projects are organized.

It changes how C++ projects are consumed.

Moreover, as mentioned, they solve a bunch of issues that have been around for many releases.

Note: One important caveat about modules: although it is, and will be, an important feature going forward, not all compilers have fully implemented it yet (at the time this book is written). The Microsoft compiler says it is mostly done, but unfortunately the gcc and Clang compilers only state as being

partially done. You need to be careful and do some investigation for the compiler you are currently using. Most of this chapter was compiled using the Microsoft C++ compiler, so at least we can show you how a lot of it works.

Before we go on, let us look at a module:

```
#include

Module10_1_1;

{



{

<< << __FILE__ << " : " << << __FUNCTION__ <<

<< "Welcome to

}

};
```

Some parts look familiar, but right at the top, we see a new keyword. The module specifier is the first line, which means that this will be a module file. Different compilers are using various filename extensions for modules, and so there is no set rule on how to indicate a module aside from this declaration. Our code gives the following output:

```
Examples in: D:\BPBPublishing_C++23\BookSamples\Chapter10.cpp

D:\BPBPublishing_C++23\BookSamples\Chapter10_1_1.ixx :
Chapter10_1_1::GiveWelcome
Welcome to C++23
```

Figure 10.1: Output from GiveWelcome

We have occasionally included in our sample code the cout statement that shows the name of the function. We do not always include it in the screenshot of the output because you know what function we are looking at, and those statements are more for when you run all the chapter examples together, so you know which output you are looking at. In this case, we do not only include the function name, but the file name as well – since in this case it seems helpful to know what file we are looking at. As you can see in the output for 10.1 above, we are running the code from and calling the function in Chapter10_1_1.ixx (the curiously named extension used by Microsoft for module files).

Getting back to the code example above the module declaration, we have and then we have our export module line, naming our module. When importing modules, there are no quotes, when defining or consuming it, and so we do not have to worry if it is quoted or bracketed. A module name follows existing naming rules, where it can contain any letter or number, but modules can also contain a dot A name with dots does not have any specific meaning, except for keeping things organized. The name could be something like we see in our next code example:

```
#include
```

```
// even if from another module, the same class cannot be exported like this

//export class Chapter10_1_1 {

//public:

// static void GiveWelcome() {



//  std::cout << std::endl << __FILE__ << " : " << std::endl <<
__FUNCTION__ << std::endl;

//  std::cout << «Welcome to C++23\n";

// }

//};

{



{

<< << __FILE__ << " : " << << __FUNCTION__ <<

<< "Welcome to

}
```

```
};

{


{

<< << __FILE__ << " : " << << __FUNCTION__ <<

<< "Welcome to C++23

}

};
```

And this gives our output:

```
Examples in: D:\BPBPublishing_C++23\BookSamples\Chapter10.cpp

D:\BPBPublishing_C++23\BookSamples\Chapter10_1_1.ixx :
Chapter10_1_1::GiveWelcome
Welcome to C++23

D:\BPBPublishing_C++23\BookSamples\Chapter10_1_2.ixx :
Chapter10_1_2b::GiveWelcomeAgain
Welcome to C++23 again
```

Figure 10.2: Output from GiveWelcomeAgain

It shows a few things:

The module name not relate to our directory folder structure.

We can export multiple classes from the same module file, although we would still want to adhere to existing coding guidelines. Normally, you would want to have your classes defined in different files, but this is just for demonstration.

You also see some commented-out code. Even though from a different module, we cannot have the same function of the same class as we exported in another module file. There is a related topic about partitions that we will learn about later in this chapter, but it makes sense that you cannot have the same class's function.

As mentioned, a longer module name with dots hints at a hierarchy, but does not relate to any type of directory folder structure. It is only for keeping things internally organized. Again, the module name does not have any forced relationship between itself and the namespace name, the file name, or the folder name in which it sits. You should still follow your conventional naming guidelines, but there is no relationship between them.

Let us quickly look at our calling code, to see what we need to do to call the modules:

#include

import

import

```
{

<< "Examples in: " << __FILE__ <<



}
```

Aside from the import statements, everything else looks familiar. To import a module, all we have to do is specify which module we want to import. After we do that, we have access to the classes and structures that we would expect.

Looking at the export lines in the previous code example again, we have multiple class exports in the same file. The point though, is that the module contains the implementation, without a header giving the forward declaration that we are used to. As mentioned, you can still have your declarations in one file, and the implementation in another. It might be a way to transition into modules, or there may be design reasons why you would want to keep them separate. Unless there is a reason why you want to keep them separate, it does make more sense to combine them, for the same reasons why we said it is not always great to separate code into a regular header and CPP file. When you decide to have separate files for declarations and implementation, there are rules that need to be followed. One is that only the module declaration file can have export statements. Speaking of exporting, near the top of our file, when it states the module will be exported, it does not mean that everything in the file will be exported. That is indicated by the line further below it, where it says we are exporting the class, which will export everything public within in. There are many ways, or scope that you can export an entity:

```
import

// exports everything withing the block; could be classes, namespaces, functions, etc.
{

}
};
{

}
};
namespace NameSpace1 {
{

}
```

```
  };

{


{}

  };

}

#ifndef _MSC_VER

// Microsoft compiler not exporting static methods yet

{}


#endif

}

NameSpace2 { // declares a namespace, and everything within the block is exported

{
```

```
{}

};

{


{}

};

}

namespace {

{


{}

};

}

#ifndef _MSC_VER

// Microsoft compiler not exporting static methods yet
```

```
{} // function is exported

#endif

{}
```

The first code block shows that everything within the block is exported. Note the NameSpace2::NotExported class is still in but it was not included in the export block. Therefore, it is not exported. Notice at the end of the file that one function is exported (or should be), and one is not, which is fine since it could still be used within the module, but not for public consumption. At the time of writing, there was an issue with the Microsoft compiler that did not like standalone functions being exported. At some point, we will assume they will fix this. Meanwhile, you will have to plan differently, and put your functions in a class or structure (which seems like a good idea to do anyway).

Just to finish things off for now, and to preview our next topic, we should mention how we brought into use. After seeing the #include statement for years, you may not have noticed that this was actually an import statement. We can use #include and import statements in the same file, although we will be using import a lot more in the future.

Recipe 10.2: What the compiler does with a module

Problem: We have learned what a module is, but how is it better?

Solution: The first answer is speed, plus helps with many issues.

Although the vast focus on this book is on coding, modules are a dramatic enough change that it would be good to know a bit more about them. We already mentioned that when a header file gets included, it is basically copying all the header text into the destination file. We also said that macro definitions can cause major headaches. You can still have templates and definitions in a module, but the definitions in a module never "escapes" out of the module. This is not a complete sequence: but a simplified overview would be as follows:

Look to see what header files that modules import can be precompiled.

Figure out the sequence (as best it can) for the modules to be built.

Compile the modules, and this is needed only once. The output are temporary files called Compiled Module Interface files, although they may have a different extension. Some older references might refer to them as Binary Module Interface but that term was changed to stress the point that these are not distributable. CMI files are built based on the compiler flags, for that particular build of the compiler, on that specific build operating system. They cannot be transported to another system (unless it was the exact same, but the effort and risk do not have much reward).

Compile the CPP files, referencing the CMI files. The result is still your traditional object files.

Build your executable, based on the CMI and object files.

The speed increase with modules has been reported by many to be threefold. That is something that will lure many into using modules, regardless of all their other benefits.

It must be stressed that the CMI files are not distributable, but they are hoping that they can be used for other purposes. Although you currently cannot parse a CMI into anything else tangibly helpful (such as an interface for consumption in other languages, or code documentation systems, and so on), people are working to see what tools they can make to leverage the knowledge held within the CMI files, and so that is something to look forward to in the future.

Recipe 10.3: How to use a module

Problem: Now that we know how to create a module, how can we use them?

Solution: We will see when we exported code from a module, we can import it into other code to use it.

We left off by saying that we could import a standard header into our module. This is called header importation. It is not a trick; what the compiler does when it sees you are trying to import a header, is to see if it can pre-compile the header (of course, after checking if it has already done so). The compiler might even see a #include statement and check if it can treat that as an import statement. If the compiler can build the header, then it will do so and include that in the compiling process. If it deems that it cannot pre-compile it, then it falls back to the normal header textual inclusion (that is, copy and paste the contents into the target file). To include a module that you created into your CPP file, you would also use the import statement. If we have our source module as:

```
import

{

{
```

```
{

<< << __FILE__ << " : " << << __FUNCTION__ <<

<< "Welcome to

SaySomethingNice();


  }


{

<< << __FILE__ << " : " << << __FUNCTION__ <<

<< "Hope you are having a good

  }

};

}
```

And our calling code would be:

```
//#include // only needed if we have output in this file
```

```
import

import

import

{

WelcomeNS::Chapter10_3_1 chapter;

chapter.GiveWelcome();

}


{

// std::cout << «Examples in: « << __FILE__ <<



SimpleImportWelcome();

}
```

Note in our calling code that we have temporarily commented out the output statement telling us the filename, so we do not have to include the header. It is not used in our calling code, and the module has already processed it in the module code. The output naturally is as the following figure:

```
D:\BPBPublishing_C++23\BookSamples\Chapter10_3_1.ixx :
WelcomeNS::Chapter10_3_1::GiveWelcome
Welcome to C++23

D:\BPBPublishing_C++23\BookSamples\Chapter10_3_1.ixx :
WelcomeNS::Chapter10_3_1::SaySomethingNice
Hope you are having a good day!
```

Figure 10.3: Output from SimpleImportWelcome

You will see that aside from the import statement being different, the code behaves as you would expect if it had included a header.

Note: A reminder that we are using the Microsoft compiler for most of the examples in this chapter, which is mostly feature-complete in terms of modules. You may have issues if attempting with the gcc or clang compiler.

If we created a new class:

```
import

import

{
```

```
: {



override {

<< << __FILE__ << " : " << << __FUNCTION__ <<

<< "Looking forward to seeing you

  }

};

}
```

And then called the function override:

```
// #include // only needed if we have output in this file

import

import

import

import

{
```

```
WelcomeNS::Chapter10_3_1 chapter;

chapter.GiveWelcome();

}

{

WelcomeNS::Chapter10_3_2 chapter;

chapter.GiveWelcome();

}


{

// std::cout << «Examples in: « << __FILE__ <<




SimpleImportWelcome();

DerivedImportWelcome();

}
```

We would get the output from the override, as it would occur if it had included a header, as shown:

```
D:\BPBPublishing_C++23\BookSamples\Chapter10_3_1.ixx :
WelcomeNS::Chapter10_3_1::GiveWelcome
Welcome to C++23

D:\BPBPublishing_C++23\BookSamples\Chapter10_3_2.ixx :
WelcomeNS::Chapter10_3_2::SaySomethingNice
Looking forward to seeing you again!
```

Figure 10.4: Output from DesrivedImportWelcome

We snuck in another change in our module code, just to see if you would notice. For we stated it was a module on the first line and then exported the module. For we skipped a step and just had the export module statement. This works, as long as it is the first line of the file. Either way is acceptable, but it is nice to know you can have one fewer line of code.

We mentioned earlier that modules do not have anything to do with other naming or organization. To show this, let us go back to our last code example from section 3 of this chapter, where we created several more entities in the module (not a good coding guideline to follow, but just to illustrate the point). As long as we have proper access to what we exported, then we can use them as normal:

import

```
import

import

{



{

<< << __FILE__ << " : " << << __FUNCTION__ <<

#ifndef _MSC_VER

PlainNothing();

#endif

  Class10_1_3a



  c10_1_3d;



  c10_1_3e;
```

```
#ifndef _MSC_VER

VeryPlainNothing();

#endif

// cannot call things that are not exported

// NameSpace2::NotExported bad1;

// AlsoPlainButNotExported();

}

};

{

: {


override {


<< << __FILE__ << " : " << << __FUNCTION__ <<

<< "It is interesting to see how calling functions is similar,\nyet it is different to include
```

```
  }


};

}
```

Just as before, we can have multiple namespace blocks with the same name. However, as before, we still cannot split a class into two areas of declaration. We can have multiple export blocks within a module, but we cannot have multiple files by the same module name. All module interface exports have to be within the same file. Since the Nothing functions did nothing, we do not have much output. We can though, see that it calls our helper function, even though that was not exported.

```
D:\BPBPublishing_C++23\BookSamples\Chapter10_3_3.ixx :
WelcomeNS::Chapter10_3_3::SaySomethingNice
It is interesting to see how calling functions is similar,
yet it is different to include them

D:\BPBPublishing_C++23\BookSamples\Chapter10_3_3.ixx :
Chapter10_3_3_Helper::CallingManyImportedFunctions
```

Figure 10.5: Output from CallingManyImportedFunctions

Recipe 10.4: Visibility and reachability

Problem: What happens when something is used in a module, but not exported?

Solution: We will see that things still partially work.

In the last topic, we talked about how the consuming code cannot see something unless it is exported. Let us look at a simple module:

```
{


const {

+

}


{}

};
```

```
input) {


result.MyValue = input;


}
```

Here we have a module that contains a struct, and a function that returns that struct. Note that we exported our function, but we forgot to export our struct. If we look at our calling code:

```
import

import


import

import

// previous code being called in other modules

import

{
```

```
// = CreateMyStruct(123);

auto result =

= result.MyValue;

= result.AddOne();

}
```

You will see that curiously enough, we can call the function, even though we did not export the resulting struct type. We can also access the member variable, and call the member function. Why are we allowed to do this?

The answer relates to something called visibility and reachability. It probably originated as a compromise when they were designing this feature, but it makes some sense. If we can export a function that returns a type, then we should be able to at least see the public members of that type, despite not knowing what that type is. You will see that when we try to refer directly to that type, we get a compiler error. That means that if not exported, it is reachable (meaning it is within our scope and we can still access it) but it is not visible (meaning we cannot name that type). In those cases, you should use auto to define your variable. We also cannot call the function obviously because that is private. It is difficult to say how useful this feature is, it feels a bit like cheating. On the other hand, it is almost like an interface that other languages have, where we do not know the type that contains the functions, yet we can still execute them.

One other little bonus, you can see in our code example above. We have our import statements as we had before, and then we have our functions that we called code that is in other modules, and then we have another import statement. Some people might not like this style of not having all the imports at the top of the file, but it is an interesting option that we can have the import statement close to where we call the other module, keeping things together.

Recipe 10.5: Layout of a module

Problem: We have looked at the module and seen some rules about them. Are there any more rules?

Solution: Some rules need to be followed, but there are also other benefits as well.

We have created and consumed a few modules now, but let us go back and look at the module parts:

```
// about to talk about our module

// and its components



// \/ \/ \/ \/ \/ \/ \/ \/

//

// global module fragment

//

#define CanHaveDefineThatIsSeenOnlyInThisModule 123
```

```
#include

//

// /\ /\ /\ /\ /\ /\ /\.



// \/ \/ \/ \/ \/ \/ \/ \/

//

// module preamble

//

import

import



//

// /\ /\ /\ /\ /\ /\ /\.



// \/ \/ \/ \/ \/ \/ \/ \/
```

```
//

// module purview

//

{

{


const { }

};

}


//

// /\ /\ /\ /\ /\ /\ /\ /\.

// \/ \/ \/ \/ \/ \/ \/ \/

//
```

```
// private module fragment

//

// /\ /\ /\ /\ /\ /\ /\ /\.



{

// ...

};

{

<< "C++23 is

}
```

As in our first example, our first line (of code, we can have leading comments) declares this is a module. In our previous example, you can see we started with export module You are allowed to combine the module declaration and the named export lines together, but you should only do that if you do not include any headers.

The header includes (along with macro defines) should go in the next section, which is called the global module fragment. You should probably

not have any include headers in any area after where you name your module (for example, export module because you could run into some issues and you would need to be very careful.

After the export module statement, is the module preamble, where you import other modules into your code. Our module code shows a great feature with export import that you can immediately export something and import it at the same time. That means that the content you imported in the module will also be accessible when you import your current module. This allows you to combine multiple smaller modules and export them together and give the consumer easier access.

Next is the module purview area, where you have your declarations (if it is a separate module interface file), or your code if it is a combined, or a module implementation file. Thankfully in most cases, the CPP code would be able to be placed here without issue.

Lastly is the section called the private module fragment. This is optional, but it shows a powerful feature. Anything in this fragment will not be reachable (or nameable). As our example shows, we can export a function (such as MyFunctionWithHidden but currently the consumer will not be able to call it.

Note: Although at one point the Microsoft compiler was able to call it, so it is difficult to say whether the current behavior is a defect, or the previous behavior was a defect.

We can still use the feature we learned in the last section on reachability, which hides the behavior. This means that you no longer need to make

extra effort with your secret implementation functions to hide the functionality, because you can allow people to run the function, but not see the function. As encapsulation is an important facet of an object-oriented language, and C++ was created to be an object-oriented version of C, you can say that it did pretty well, but adding modules helps it do an even better job. C++ classes do allow you to group code and variables, but allowing consumers to see some of the details via the header file was far from ideal. With the private module fragment, we can now easily hide the implementation from the outside world (that is, the calling code). This means that the code will look cleaner without the extra tricks and extra implementation methods. Moreover, it is obvious what the intent is if it is in the private module fragment.

When we look at our calling code:

```
//import Module10_1_1;

//static void SimpleImportWelcome() {

// WelcomeNS::Chapter10_3_1 chapter;

// chapter.GiveWelcome();

//}

// ...



{
```

```
MyModuleNS::MyClass c1;


int x =

// MyFunctionWithHiddenImplementation();

WelcomeNS::Chapter10_3_1 chapter;

chapter.GiveWelcome();

}
```

You can see in LookAtOverview() that we are trying to call our MyFunctionWithHiddenImpementation() function, but it said it was not found.

Also note that despite the import WelcomeModule; commented out, we have access to what it exported because we had done an export import statement for The SimpleImportWelcome() code was left in this code snippet, to show that if you uncomment out those lines, there is no issues with duplicated defined entries. The compiler understands the same module is being imported twice, even if one is indirect.

Recipe 10.6: Cyclical modules

Problem: Since it could be a problem with cyclical dependencies when including headers, how does it work with modules?

Solution: There is a solution, but there is some work involved.

Modules offer incredible power, but nobody is saying the switch is zero effort. One good example is when you have a cyclical header includes. Typical with headers, that means when file1.h includes and file2.h includes That issue would be greatly complicated if other macro defines were involved, but even without them, compilers have to be smart (which they are; with pragmas and other possible header guards) to avoid problems.

When compilers look at cyclical headers, they try to do the best they can, and if it does not work, then you need to perform some tricks or modify your source code. If there was a chance that there were cyclical modules, this time the compiler will just say it will not even try to resolve it, and you need to modify the code.

If you had one module that had a forward declaration:

```
#ifdef DOES_NOT_COMPILE
```

```
import

SecondType* pSecond; };

#endif
```

And another module that had the opposite forward declaration:

```
#ifdef DOES_NOT_COMPILE

import

FirstStruct* pFirst; };

#endif
```

The compiler will just fail, with some error that basically says you need to fix your bad design. The compromise to be made, is to combine them into one module, and have a forward declaration:

```
FirstStruct* pFirst; };
```

```
SecondType* pSecond; };
```

This is less than ideal and could cause problems in large, complex systems, but one of the few shortcomings with modules.

Recipe 10.7: Importing the standard modules.

Problem: We want to have to include fewer headers when dealing with modules.

Solution: We will learn in C++23 that there are two standard modules: std and

We saw earlier in this chapter that it was easy to have export import statements, to amalgamate other modules into one. That is what was done in C++23 to make it easier for developers to have cleaner code, and simplify the coding import process. They defined two modules: std and

The std module includes:

Everything in the std namespace from the C++ library headers that was capable of being exported.

The C++ headers for C library facilities (such as std::puts() from

Declarations in the global namespace for storage allocation and deallocation (such as the new operator).

The std.compat module includes:

Everything mentioned in the std module

Declarations in the global namespace corresponding to declarations in the std namespace that are provided by C++ for the C library (such as the fclose() function)

This means that you should be able to reduce the number of import statements required for some basic functionality:

{

{

<< << __FILE__ << " : " << << <<

message = "Hello

<< message <<

}

};

For more complex code, you will probably still have to include others, but this helps simplify things for some of the simple code files.

Note that currently the Visual Studio compiler names the module but at this current time it is unknown if this will be the final name.

Also note, that as the time of writing, Microsoft has not fully implemented the import std.compat module yet. They say there is a workaround, but we will not bother including it, and hope it gets resolved soon.

Recipe 10.8: What are module partitions

Problem: We would like a mechanism to split a module into multiple files.

Solution: We will learn about module partition units.

As mentioned earlier, a module's name follows standard conventions, based on letters, numbers, and dots. There is no hierarchy involved. It is possible that you can have a module split into multiple implementation files, and this may be beneficial if your module becomes too large, or you want to separate the code into logical pieces. The mechanism for dividing up a module is called partitions.

We have learned how you can immediately export the module you are importing. This is similar but has the extra benefit that it knows that other modules are parts of a larger module.

We have divided our code into a couple of modules. There is no rule about how to partition them. It can be done with type definitions versus implementations, or into subcomponents, or however, you think makes sense.

};

Our other module file contains:

```
import

};

left, RightHandType& right) {

// ...

}
```

Note the new character in the module name. The colon) denotes it as a partition, and so it knows that Left and Right are partitions in the Chapter10_PartitionExample module. You will also notice that when we import a module, the name starts with a colon. This tells the compiler that it is a sibling partition, so we do not have to specify the full name, but just the ending partition part. And in our module, combines them:

```
Module10_8_PartitionExample;



right, LeftHandType& left) {

// ...
```

```
}
```

Here you can see again that we can export and import at the same time, but the name is specified with a colon, indicating that it is a partition of the parent.

This partitioning mechanism is another feature that allows you to organize your code more cleanly and easily than if you only had header files to use.

Conclusion

Hopefully, you can understand how modules are a great addition to the C++ language, and how impactful it may become. Remember you have lots of choices on how to implement them. You can have separate declaration and implementation files as we have traditionally had, or you can combine them to avoid duplication. You could rewrite your code to get rid of the secret implementation functions you created, and leverage the private module fragment to hide them that way. You can also split your code into module partitions, to organize things logically that way. Once you migrate to modules, you will notice an improve in your compile times and appreciate that your code looks cleaner.

Points to remember

Modules were a huge new feature of C++20, and there could be ripple effects on how they are used for years to come.

Module names are not related to namespaces, class/type names, or file/folder names, but you should carefully think about the logical hierarchy of your naming convention.

Although modules are great for new code, it will take effort to improve your old code.

The private module fragment should greatly improve encapsulation, as well as readability, or your code.

You can import the std or std.compat modules, to simplify your import (or include) statements. If you need other headers, you should try import rather than including them.

Partitions are a handy way of adding an extra logical layer of organization.

References

CoreCPP. (n.d.). CoreCPPModules.

Wikipedia. (n.d.)

Join our book’s Discord space

Join the book’s Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 11

Introduction to Ranges and Views

Introduction

Another entry for C++ 20's Big ranges are an extremely handy way to deal with data. They will simplify, reduce and speed up your code. Quite a lot of existing C++ library code has new overloads to accept ranges, which means that your code will not need to specify an iterator any more since the range will suffice.

Structure

In this chapter, we will cover the following topics:

Recipe 11.1: What is a range

Recipe 11.2: What is a view

Recipe 11.3: What is a span

Recipe 11.4: Range concepts

Recipe 11.5: Range primitives

Recipe 11.6: What is a subrange

Recipe 11.7: What is ranges::dangling

Recipe 11.8: What is a view_interface

Recipe 11.9: What is an owning_view

The great news is that it seems the three major compiler makers are C++23 feature-ready for the recipes in this chapter.

{

// works with compiler? when tested clang

// 20231031 yes

// 20231031 yes yes yes

// 20231031 yes yes yes

// 20231031 yes

// 20231031 yes

// 20231031 yes

/* do not need to run examples for

* section 7, for viewing only */

// 20231031 yes

// 20231031 yes yes yes

// 20231031 yes

```
// 20231031  yes  yes  yes

}
```

Objectives

Ranges and views are another big leap in the C++ language. Similar to when templated containers were first released. It was not a question of understanding how to use them, but there was so much new functionality you needed to learn the best way to use them. By the end of this chapter, the reader will understand about what ranges and views are, and a bit about how to use them.

Recipe 11.1: What is a range

Problem: We want a simpler way of working with containers.

Solution: C++ 20 has a lot of new functionality with ranges.

Before we talk about ranges, we should mention a few other things first. Standard library containers (such as vector, map, and queue) have been a mainstay in C++ for several releases). Having templated containers, meant you could iterate and access values for various types of collections in a similar manner. When C++20 came out with concepts, it was a mechanism to define iterators, so you could tell that the iterator was forward, bi-directional, random-access or something else. It also gave the concept of a sentinel, so that it was more of a state indicating whether the iterator was done, versus explicitly stating it had gone past the last element for the container.

We can combine some of those ideas and modify functions in the standard library to make the parameters simpler. Instead of passing in two parameters, one iterator defining the start, and another iterator defining the end, we could pass in just one parameter. The one parameter would be something new, which would define the starting and ending iterators, or it could hold the start and a sentinel. This new entity is what ranges are all about. Although it is not just about passing one less parameter to an existing function. It is also about having many new functions that allow a more functional way of programming. However, we will talk more on that a bit later. We can rethink that any of our standard library collection

classes are ranges, although containing all values. We can define other ranges to be a portion of that complete collection.

There are four different ways to define a range:

To provide a begin and end iterator, defined as [being, end)

For conditionally-terminated sequences, defined as [start, predicate)

As a counted sequence, defined as [start, size)

And for unbounded sequences, defined as [start..)

The first two range-related headers to mention are which defines the basics, as well as some range-related enhancements to

Let us look at a simple example:

```
#include "Chapter11.h"

#include

#include

#include
```

```
#include

#include

#include

{

numbers{ 55 };

numbersBad{ 55 };

auto answerGood = numbers.end(),

<< "Found good answer " << * answerGood <<

auto answerBad = numbersBad.end(),

if == numbersBad.end()) {

<< "Bad answer not found" <<

}


else {

<< <<
```

```
}

auto answerGood2 =

<< "Found good answer in range " << *answerGood2 <<

auto answerBad2 =

if (answerBad2 == numbersBad.end()) {

<< "Bad answer not found in range" <<

}

else {

<< "Found bad answer in range" << *answerBad2 <<

}

}


{

FindInRange();
```

```
}
```

Unsurprisingly, the output is as follows:

```
Found good answer 42
Bad answer not found
Found good answer in range 42
Bad answer not found in range
```

Figure 11.1: Output from FindInRange

We will talk about all the functions available later but notice how much cleaner it looks when you can just pass in the collection to a function.

There are a couple of tricky parts to keep in mind. For the most part, ranges are non-owning. It means that the collection the range was looking at or created from (depending on how you like to word it) still owns the data. The range is similar to an iterator (which makes sense, since a range knows of at least one iterator), in that it is just looking at the data in the collection. Another tricky part when learning about ranges is differentiating between a range and a view. We will talk about views next.

Recipe 11.2: What is a view

Problem: We would like a lightweight wrapper for a container.

Solution: Related to ranges, C++ 20 also has views.

The first thing we probably think of for a view and C++, is a This C++17 class template relates to a sequence of characters. Similar to a string, we can iterate through it, get the value, and get the length of it. The main difference between a string_view and a string, is that the string_view does not own the contents. This is similar to the range's view. Similar to what we talked about in the last topic, a std::string is like a where those are ranges. On the other hand, a string_view is just a view. A view is a range but with extra restrictions. The main goal of a view is to be lightweight. A quick example would be:

s) {

<< s <<

}

s) {

<< s <<

```
}

{

s = "Hello

PrintString(s);

PrintStringView(s);

}
```

We are not passing by reference, and so when we pass the string to it creates a copy, which could take a while and waste memory. When we pass the string to it creates a view but does not create a copy of the content. Copying a view should always be quick because it does not modify the content.

Another differentiator is eager versus lazy functions. In the ranges library, we have range algorithms, which are functions that are applied to a range eagerly. Eagerly meaning the function acts right away on the range content. The ranges::find function, that we used in the last topic example, acted on the range right away. In the ranges library, there are also range adaptors, which act on a view lazily. Lazily implies they will not act until required, and that too is normally when it is getting referenced.

Here is another example looking at a view:

```
view) {

for : view) {

<< n << ",

}

<<

}

{

numbers{ 55 };

PrintView(numbers);




}
```

Here we can see that we can print the whole container of integers, since we can say that a vector is a range, and a range is a view. We can also create a view that contains all the numbers in our container, or we can even use offsets of the begin and end iterators. We pass those into ranges::subrange which creates a view we can use. We are cheating a bit

using the auto keyword in PrintView for the parameter type, but there is no quick and clean way to define the view. We will get into the definitions and differences more as we go through this chapter. You can think of a view as a lightweight wrapper of the actual container/object that holds the data. Most times, the view is not the owner of the data, and because of that, it is easy to quickly create or copy a view.

A little side note, or word of caution. They may have fixed some issues with views in C++23, or will in C++26, but in C++20, there were quite a few problems with views ignoring const-ness.

Recipe 11.3: What is a span

Problem: We would like a lightweight wrapper for a container that knows its size.

Solution: C++ 20 was released with

C++20 also has a templated class called which could have easily fit in our upcoming chapter on containers. We are talking about it now because it has some similar traits to range and view, and so it seems worthwhile to differentiate them. Similar to view, spans are lightweight because they do not own the data they are looking at. You can iterate a span and get the size of it. One place that you will see span used a lot is as a function argument. If you pass an array to a function, traditionally we would declare it as a pointer (or that is what the compiler assumes it is), but the problem then is that we do not know the number of elements in the array. A span can understand the size of an object, as we can see in this example:

```
span) {

<< "Number of items= " << span.size() <<

for : span) {

<< i << "
```

```
}

<<

}

{

numbersVector{ 55 };

PrintSpan(numbersVector);

67 };

PrintSpan(numberArray);


}
```

This gives us the following output:

```
Number of items= 5
1 5 17 42 55
Number of items= 6
1 5 17 42 55 67
```

Figure 11.2: Output from CreateSpanToPrint

We will go back to std::span again in the collections chapter, but it seemed important enough to distinguish that although std::span shows similarities to std::ranges and views, they are for different areas. Refer to Table

Table 11.1: Differentiating between a range, a view, and a span

Recipe 11.4: Range concepts

Problem: It will be good to have a variety of concepts to describe ranges.

Solution: There are quite a few included in the C++20 release.

We will be repeating ourselves when we stress how big concepts were in the C++20 release because here we are talking about how good of a feature ranges are, but ranges are basically just concepts. Based on those new concepts, a lot of other functionality can be created and extended.

The basic ranges::range is a templated concept, and is defined by two functions:

This returns a constexpr We have already talked about and we have talked about the iterator. It is interesting how these pieces come together to make something.

This returns something different, a In that way, it relates to an iterator, but we mentioned that a sentinel is not always going to be just past the last item. It may relate to an indicator of whether you are done.

Based on the inference whether the compiler sees those functions on an object, it is determined whether you have a range or not.

The next concept to mention is a This has the same constraints as but it also has a size() function. A simple example would be:

```
auto range) {

<< "The size is " << range.size() <<

}

{

numbersVector{ 55 };

LookAtSizedRange(numbersVector);

}
```

Another refinement of is This specifies that begin() and end() return the same type. Remember that if you can specify, a function is a requirement of a concept without specifying the return type. This concept clarifies that requirement by saying they need to return the same type. You would normally have a different return type for the end function (often a Boolean) when dealing with a sentinel flag.

We will take a look at the next group of concepts altogether. They should sound familiar to you, since they are related to the iterator concepts we talked about in a previous chapter. Refer to the following table:

table:
table: table: table: table: table: table: table: table: table: table: table: table: table: table: table: table: table: table: table:

table: table: table: table: table: table: table: table: table: table: table: table: table: table: table: table: table: table: table:
table: table:
table: table:
table: table:
table: table:

Table 11.2: Describing the iterator-related range concepts

The next concept to talk about is A view is a refinement of but also requires that it is We covered that in an earlier chapter, and it means that it can be move constructed, move assigned, or l-values can be swapped. The

view concept also has the requirement it derives from but since we never create a view by hand, we do not know, and do not need to know, what that is.

Speaking of views, there is also the concept This is a refinement of range that specifies the range can be converted into a view (via an adaptor, which we will talk about in an upcoming chapter).

The next concept is called which is a refinement of and simply says it returns a constant iterator.

The last concept is which has the requirement of being able to order the range’s items in a particular sequence. The most used way to do this is which we will learn in a upcoming chapter.

Recipe 11.5: Range primitives

Problem: We want to know if there are any new primitives.

Solution: C++ 20 has several new range-related primitives.

We do not have to spend a lot of time on this, because you would have seen similar types before. It is nice to know the primitives, for those cases where you might need to specify them, instead of just using the auto keyword. They are all in the std::ranges namespace, with T being the type in the container, and R being the range type. Refer to the following:

is the iterator type (of type T).

is the constant iterator type (of the range type R).

sentinel_t and is the sentinel type (of the range type R), and the latter being constant.

is the size type of sized_range type R.

the difference type of the iterator type.

is the value type in the range.

range_reference_t and is the reference type of the iterator.

is the rvalue_reference type of the iterator.

is the common_reference type of the iterator.

Most of these have similar counterparts in the iterator category, or just in the standard library.

Recipe 11.6: What is a subrange

Problem: We would like an easy way to obtain part of a range.

Solution: In the C++ 20 ranges library, there is

We have seen many times with C++, that there are several ways to perform a similar task. To get a subset of a range, you could define your begin and end iterator markers, or you end sentinel, or create a span, or even create a coroutine that yielded the values you wanted to look at. They also created a new templated class called

There are a number of different ways to create a subrange, and the returning value is a view. As you would expect, you could specify begin and end iterators, or a begin and a sentinel. From there, we can iterate and get the value as you would expect, see if it is plus a few other things, as we can see in our sample:

```
{

numbersVector{ 255 };

<< "All of vector:

for it = numbersVector.begin();
```

```
it != numbersVector.end(); it++) {

<< *it << "

}

<< <<

= numbersVector.begin() +=

= numbersVector.end() -=

auto sub = endingIt);



<< "making subrange of third to second last, size="

  << sub.size() << ":

for it = sub.begin(); it != sub.end(); it++) {

<< *it << "

}

<<

<< "subrange (front-end) is " << sub.front() << "-" << sub.back()
```

```
  << <<

=

<< "partial subrange after advancing 3, size="

  << partialSub.size() << ":

for it = partialSub.begin(); it != partialSub.end(); it++) {

<< *it << "

}

<< <<

partialSub =

<< "partial subrange after advancing -2, size="

  << partialSub.size() << ":

for {

<< n << "

}
```

```
<< <<

<< "getting an element of the original subrange, [1]="

  << <<



}
```

Running this gives the output shown in Figure

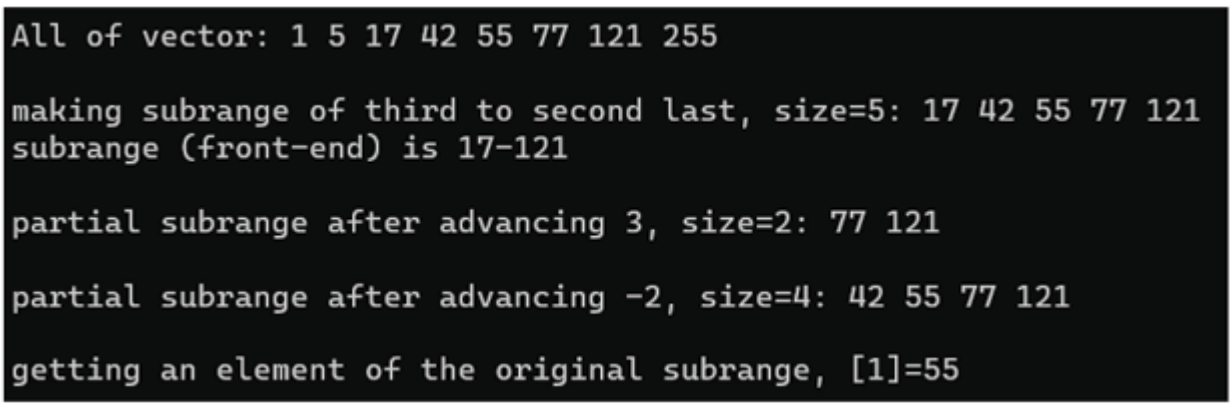
All of vector: 1 5 17 42 55 77 121 255
making subrange of third to second last, size=5: 17 42 55 77 121
subrange (front-end) is 17-121
partial subrange after advancing 3, size=2: 77 121
partial subrange after advancing -2, size=4: 42 55 77 121
getting an element of the original subrange, [1]=55

Figure 11.3: Output from LookAtSubrange

We created our subrange by defining the iterators. In our case, we just advanced the start and moved the end backwards a bit. The output shows this, as expected. We then called advance() on the subrange, and this returns a new subrange, but it keeps off where the iterator is. We then call advance() again with a negative number, and it knows it can move the iterator backwards. Keep in mind that you can do that only if your original

range is bidirectional. Finally, knowing our range allows random access, we printed out the second element of the subrange using the index [1].

Similar to front() and you could also call You would specify 0 for the iterator, and 1 for the sentinel.

Recipe 11.7: What is ranges::dangling

Problem: Is there a way to be more careful with our dangling iterators?

Solution: The ranges library has which indicates we should be careful about when using the range.

You should know what dangling pointers are, or at least experience them.

```
{

Chapter11* chapterToBeDelete = new Chapter11();

Chapter11* pChapter = chapterToBeDelete;



// pChapter is left dangling, will get exception if used

pNumber =

{

= { 67 };

pNumber =
```

```
}

// numbers has gone out of scope, pNumber is left dangling

it;

{

numbersVector{ 255 };

  it = numbersVector.begin();

}

// the vector goes out of scope, and the iterator is left dangling

}
```

The sample has shown a few different ways to cause a similar problem. If you reference something that gets deleted, the pointer still exists, but the allocation it pointed to is now gone. We do not have to point to heap allocation to have the problem. It can happen in a function (which we just have as a code block for brevity) and you return a reference. When the block (or normally function block is done, that variable is no longer allocated. A similar problem arises if you have an iterator. After the block is done, the iterator is now pointing to something that no longer exists.

Unfortunately, even the latest release does not solve this problem, or at least not totally. With ranges, the problem that often (or could often) happen is that one creates a range from a container or array that is on the stack and about to get deallocated. Once again, the range exists but does not point at valid memory. What the latest release can do, is notice what was given to create the range. If the range knows was an r-value (that will not be persisted), then on trying to use that range, via one of many range functions, it will return If you ask what a dangling_iter does, the answer is nothing. It is an empty type that does not have an indirection operator, (operator This means that if you try to use the you will get a compiler error, instead of a runtime error.

The following is an example creating a similar issue:

```
{

auto arrayThatWillCauseProblems = [] { 67 }; };



auto badIter =

if {

// std::cout << «Bad iter cannot be used" <<

auto x =

}
```

```
}
```

We are calling our lambda that returns an array, and passing it into the ranges::max_element function (which will learn more about in an upcoming chapter). The lambda returns an iterator, which initially points to the highest element of the range. If we used a function that just returned the then it would just return the value and we would not have the problem. Without max_element() function, it correctly finds the right element, but the problem is that the array becomes out of scope. The compromise for this problem, is that the function sees the type that is being passed in, and knowing it will go out of scope, and returns a dangling_iter type. If we try to use the value the iterator points at (or better phrased, pointed at), you will get a compiler error. There are many ways to fix this code, whether you allocate memory for the array, or copy the array so that it stays in scope, or it might also be possible you can return the maximum value and not an iterator. You have many choices in how to solve the problem, but at the very least, you have to be aware of how you created the problem.

Another mechanism to use as a compromise is This flag will tell your compiler that you have resolved the risk of having a It is not smart enough to ensure you have solved the problem; it is a mechanism to tell the compiler that you have solved the problem:

```
template T, N>

: N> { };

template T, N>
```

```
N>> =

{

= [] {

67 };

values };

};

if {

<< "BorrowedSpan knows how not to leave dangling iterators" <<

}

=

if == {

<< "Good iter has current value of " << *goodIter <<

}

}
```

The templated type BorrowedSpan has an added flag ranges::enable_borrowed_range set to true. There is a ranges::borrowed_range() function that we can use to verify the flag is set, and we can use ranges::max_element() to get an iterator that we can use. We solved our problem in this case by having static values, but most likely, you will have a different solution. Despite our values being static, if we did not have that flag set, then we get a similar compiler error. There are several types of views that can have this flag set as true, along with and

Recipe 11.8: What is a view_interface

Problem: We would like an easy mechanism to create a view.

Solution: In the C++ 20 release, we have

We have learned a fair amount by now about ranges, so let us talk about views. We can say a view is based on a range, but the data probably looks different than if just looking at a range. There are two main actions you would perform to get a view; you would filter the data, or transform the data. Filtering the data means you are specifying criteria so that your result is (probably) less than what you were looking at before. (The word probably is added in, because if your filter criteria are weak, then you might have all of the data pass through so that your view data will not look any different). The other way is to transform it, meaning the actual data values in the resulting view will be different.

The view_interface templated class is a base class used by the view classes in the ranges namespace. You can also use it as a base/helper class if you need to create your own views. Most times when creating a view, you would use Curiously Recurring Template Pattern

CRTP is a mechanism to define a type, and at the same time, pass the type as an argument to the base type. If we look at the example:

```
template
```

```
: {



vec) :

vbegin(vec.cbegin()),



  {}

const { }

const { return vend; }



vend;

};

{

ingredients =

{ "soy sauce" };

ingredients };
```

```
if == {

<< "There are " << vv.size() << " ingredients" <<

for it = vv.begin(); it != vv.end(); it++) {

<< *it <<

  }

<< "To repeat, they are: " <<

for : vv) {

<< i <<

  }

}

}
```

The output is shown in Figure

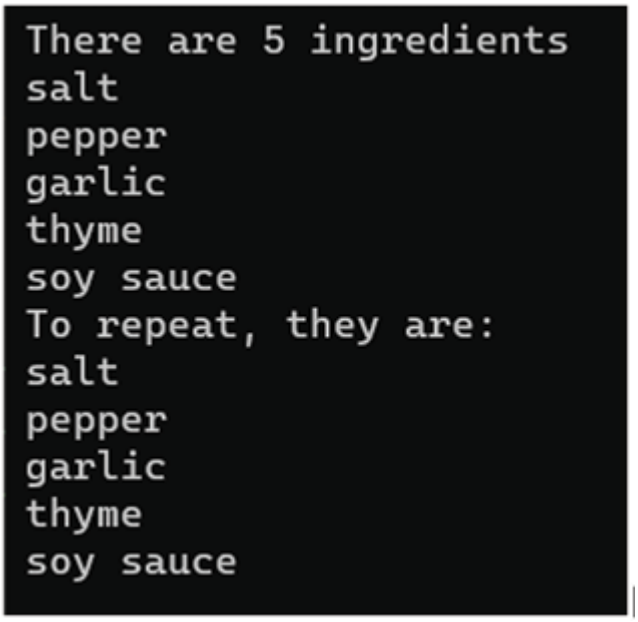

Figure 11.4: Output from CreateViewInterface

You can see that when we are defining the MyView class, we are passing in MyView as a parameter to the type In this case, it is done to inspect the capabilities of that container type. For view_interface specifically, it is looking at the capabilities of the range so that it can determine what capabilities the view has. You normally supply the begin() and end() functions, and so the view_interface can determine the rest of the implementation. Our example is not as generic as it could be, since we defined it as passing in a [Table 11.3](#) shows the potential view_interface

Table 11.3: Potential view_interfacefunctions

You might be wondering why are views so important. Why do all this work for a view, that is a wrapper around a range, which is a wrapper around a container? The complete answer sadly, will not be clear until a latter chapter on view adapters. It is about views being composable, and being able to chain them together to make cleaner code that does powerful things.

Recipe 11.9: What is an owning_view

Problem: We would like a view that has ownership of a range.

Solution: There is

Although technically a range adapter, we will just mention this view in this chapter. Partially because it does not have the pizzaz as some of the other views have, and also because it ties in with some of the topics we have mentioned in this chapter. Simply stated, ranges::owning_view derives from and owns/stores the range within it. It has the same member functions as but also has base() which returns the reference to the range. You would not use it too often by itself but could be useful in some cases when used with other adapters.

Conclusion

This was the first of the four chapters talking about ranges and views, so you can correctly assume that they are a big deal. We did not get into all the powerful features they have, but we needed to go over some of the basics first. Although some of the definitions and capabilities do overlap, they each have their benefits and strengths.

Points to remember

Ranges are based on concepts.

Ranges and views are related to each other, and to the containers they reference.

Most of the ranges and views, do not own the data they are referring to.

Although it is still possible to leave a variable dangling, the ranges library has considering that and will give you a compiler error to stop you from having bad code.

The different types of ranges, relate to the different types of iterators (and their abilities) we have looked at.

Range algorithms are lazy.

References

CppReference. (n.d.). BorrowedRange. Retrieved from https://en.cppreference.com/w/cpp/ranges/borrowed_range

YouTube. (n.d.). MeetingCPP2022. Retrieved from https://www.youtube.com/watch?v=O8HndvYNvQ4

CppReference. (n.d.). CRTP. Retrieved from https://en.cppreference.com/w/cpp/language/crtp

CppReference. (n.d.). ViewInterface. Retrieved from https://en.cppreference.com/w/cpp/ranges/view_interface

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 12

Range Access and Non-Modifying Sequence Functions for Ranges

Introduction

In the previous chapter, we got an introduction to what ranges and views are. We started to learn some of the basic functions that a range and view has. In this chapter, your knowledge of range functions will explode. We will check with the basic access functions, and dive into the dozens of range algorithm functions available to us. These eager operations (they execute their action right away) are similar to many in the normal standard library, but there are some new functions as well. It is also handy that these operations work, no matter the underlying container type. A lot of these algorithms were released when ranges first came out with C++20, but there have been some noticeable new ones in C++23.

Structure

In this chapter, we will cover the following topics:

Recipe 12.1: Review the basic range access functions

Recipe 12.2: The for_each functions

Recipe 12.3: Looking at count, any_of, all_of, and none_of functions

Recipe 12.4: Looking at comparison functions

Recipe 12.5: Search functionality

As we have been doing, here is a chart of our topics, and whether the compiler is feature-ready. In most cases, things are looking good.

{

}

Objectives

By the end of this chapter, the reader will understand a lot more about the operations that you can perform on a range. You will learn about how flexible these new functions are, and where are the best situations to use them.

Recipe 12.1: Review the basic range access functions

Problem: We want the same functions with ranges as we would have with the other standard containers.

Solution: The C++20 ranges library has the basics covered.

We have already seen a number of these functions in other examples, but let us start my stating what headers we will be using in this chapter:

```
#include

#include

#include

#include

#include

#include

#include

#include
```

```
#include

#include

#include

#include

#include

#include
```

And if we follow our code here:

```
{


title, auto r) {

using R =


=

<< "Looking at range for " << title << " / "

  << <<

  << " of " <<
```

```
  << ", size=" << << ", ssize=" <<

  << ", empty=" << <<

  <<

for it = it != it++) {

<< *it << "

  }

<< <<

}


{

<< << "Examples in: " << __FUNCTION__ <<

const myVector = { 67 };

myVector);

//

= { 67 };
```

```
= /

auto sub = cArrayCount);

style array (via sub);


//

myList = { 67.5 };

myList);

//

myQueue = { 67 };

myQueue);

//

mySet = { "you?" };

mySet);

//

// does not work with range functionality yet
```

```
//std::mapstd::string> myMap = { { '1', "Hello" }, {'2', "World"} };

//WalkTheRange("map", myMap);

}

};
```

We still have our begin() and end() functions to get iterators (or iterator and sentinel). We also have the empty() function to state if there are no items in the range. The size() and ssize() functions tell you the number of items, with the latter giving you the size as a signed type. Our output is shown in Figure

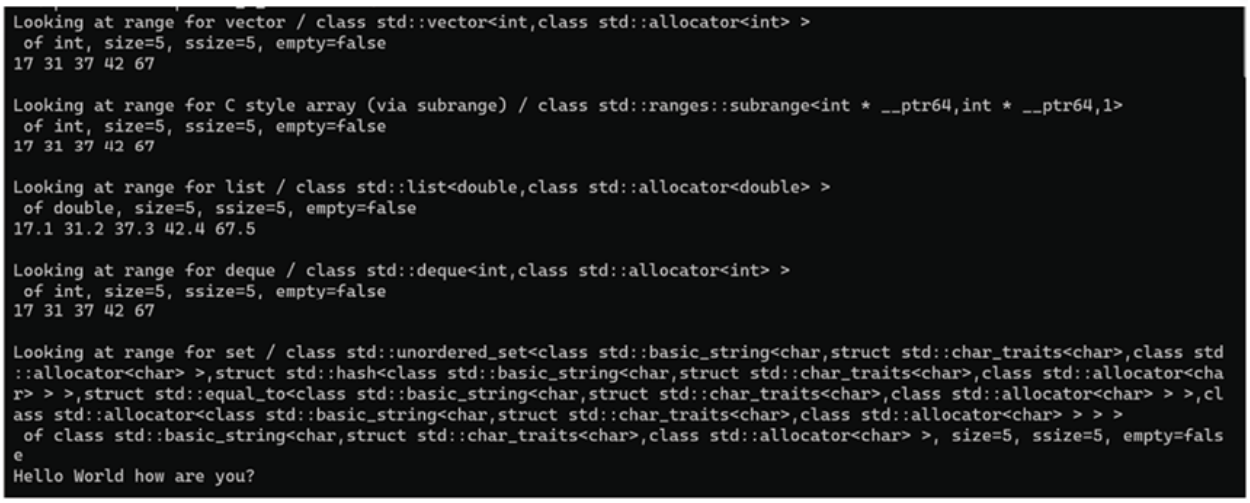
Looking at range for vector / class std::vector<int,class std::allocator<int> >
of int, size=5, ssize=5, empty=false
17 31 37 42 67
Looking at range for C style array (via subrange) / class std::ranges::subrange<int * __ptr64,int * __ptr64,1>
of int, size=5, ssize=5, empty=false
17 31 37 42 67
Looking at range for list / class std::list<double,class std::allocator<double> >
of double, size=5, ssize=5, empty=false
17.1 31.2 37.3 42.4 67.5
Looking at range for deque / class std::deque<int,class std::allocator<int> >
of int, size=5, ssize=5, empty=false
17 31 37 42 67
Looking at range for set / class std::unordered_set<class std::basic_string<char,struct std::char_traits<char>,class std
::allocator<char> >,struct std::hash<class std::basic_string<char,struct std::char_traits<char>,class std::allocator<cha
r> > >,struct std::equal_to<class std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> > >,cl
ass std::allocator<class std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> > > >
of class std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> >, size=5, ssize=5, empty=fals
e
Hello World how are you?

Figure 12.1: Output from LookAtBasics

You can see in the output that we printed out the container type and the container item type. You can also see that at times, it can get rather long and messy. One thing of note is that we had to manually convert the raw array to a range by performing a but after we did that, everything worked as well as the

other types. The only problem is that we cannot tell from the output what the input container was.

If we create another pair of functions, you can see that we can also iterate the range in reverse:

```
{



title, auto r) {

using R =

using Elem =

<< "Looking at range for " << title << " / "

  << <<

  << " of " <<

  << ", size=" << << ", ssize=" <<



  << ", empty=" << <<

  <<

for it = it != it++) {
```

```
<< *it << "

  }

<<

}


{

<< << "Examples in: " << __FUNCTION__ <<

myVector = { 67 };

myVector);

//

= { 67 };

= /

auto sub = cArrayCount);

style array (via sub);

//
```

```
myList = { 67.5 };

myList);

//

myQueue = { 67 };

myQueue);

//


mySet = { "you?" };

mySet);

}

};
```

The output is shown in Figure

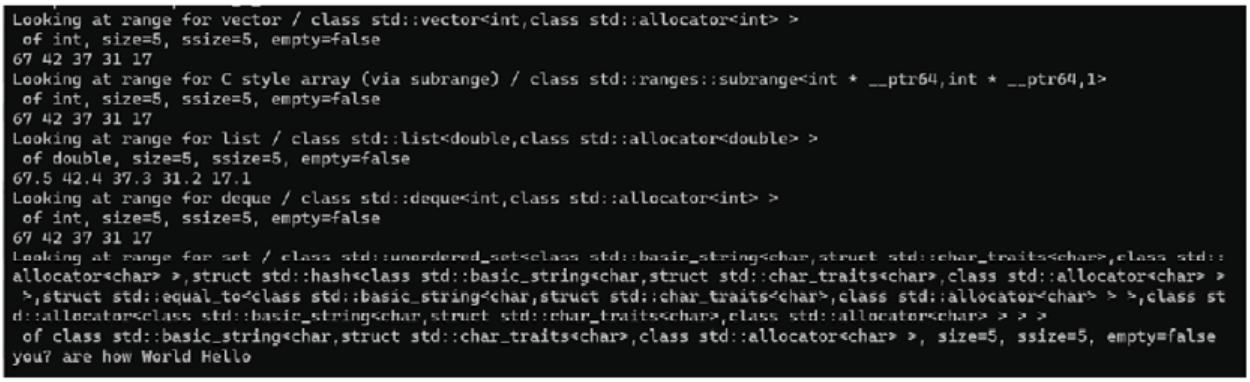
Looking at range for vector / class std::vector<int,class std::allocator<int> >
of int, size=5, ssize=5, empty=false
67 42 37 31 17
Looking at range for C style array (via subrange) / class std::ranges::subrange<int * __ptr64,int * __ptr64,1>
of int, size=5, ssize=5, empty=false
67 42 37 31 17
Looking at range for list / class std::list<double,class std::allocator<double> >
of double, size=5, ssize=5, empty=false
67.5 42.4 37.3 31.2 17.1
Looking at range for deque / class std::deque<int,class std::allocator<int> >
of int, size=5, ssize=5, empty=false
67 42 37 31 17
allocator<char> >,struct std::hash<class std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> >
>,struct std::equal_to<class std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> > >,class st
of class std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> >, size=5, ssize=5, empty=false
you? are how World Hello

Figure 12.2: Output

We are using crbegin() and which is similar to other container functions to go in reverse, and with a const iterator. There are of course cbegin() and cend() for constant-forward iteration, plus rbegin() and rend() for non-const reverse iteration.

The other couple of functions to touch upon are data() and Both return a pointer to the first element of a contiguous range of data. Again, the latter cdata() method returns a constant value as shown:

{

{

<< << "Examples in: " << __FUNCTION__ <<

input{ "Hello C++23" };

ptr =

```
// does not compile, since cannot set non-const pointer to const ptr

//char* ptr = std::ranges::cdata(input);

cptr =

<< "Input is: " << ptr <<

  << "and so is:" << cptr <<

}

};
```

The output can be seen in the following figure:

```
Input is: Hello C++23
and so is:Hello C++23
```

Figure 12.3: Output for LookAtData

Recipe 12.2: The for_each functions

Problem: We want ranges to be easy and more of a functional-style coding.

Solution: It is simple to apply a predicate to the range.

Many new range-related functions were added to the header, and many are similar to existing container functions, along with some new ones.

The first function we will talk about is Similar to the ranges function iterates through the range, invoking the function that was a supplied parameter, with each item in the range. Using the same headers as before, we have some sample code:

```
{



{

<< << "Examples in: " << __FUNCTION__ <<

myWords = { "release" };

auto PrintFn = w) { << w << };
```

```
<< "The words being printed separately:" <<

PrintFn);

<<

auto [it, output] = Append());

<< "The sentence being printed:" <<



<< output.sentence;

if (it == myWords.end()) {

<< "!" <<

  }

<<

auto sub = myWords.size());

auto [itLoud, outputLoud] = AppendLoudly());

//auto [itLoud, outputLoud] = std::ranges::for_each(myWords,
AppendLoudly());
```

```
<< "The words being printed again:" <<

PrintFn);

<< "The sentence being shouted:" <<

<< outputLoud.sentence;

if == myWords.end()) {

<< "!" <<

  }

}

};
```

In our sample here, we are looping through multiple times. The first time, we are calling a lambda named and each iteration is passing the value by const reference. Next, we call a member function (actually, it is the operator() function in Append, passing a copy of the range's item by value). Note that the returning result is a pair; containing the end iterator and the resulting Append struct. Then we call another operator() function, but this time passing a reference, so that we convert the text to uppercase. We actually do this by calling There are many ways to change the case, but it seemed interesting to show the old way on how you always had to

pass the begin() and end() as parameters. Then we print our resulting sentence again, and you can see that the input was modified, as shown in Figure

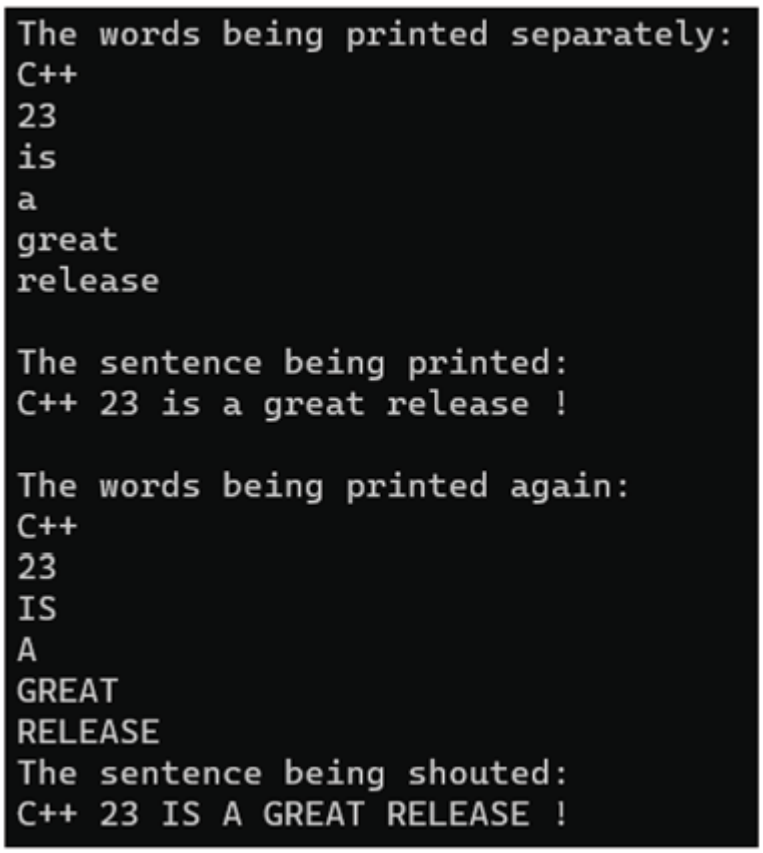

Figure 12.4: Output of DoForEach

We are outputting a fair amount of text, normally in code you would have assert statements, but in some cases, like a book, it is nice to show output. A similar function we can take a look at is This also invokes a function, but you specify how many iterations you will call it for. Let us refer to an example of this:

{

```
myQueue = { 67 };

= n) { << n << " };

= n) { n *= };

myQueue.size(), PrintFn);

<<

DoubleFn);

PrintFn);

<<

}
```

We are just using lambdas to keep things small. Note that although we are calling a range function, since we are not passing in a range, we still have to resort to calling the begin() function to pass in an iterator. The next parameter is the number of iterations to execute the function. An important thing to remember is that it does not seem (at least with the compiler used here) like it is doing any bounds checking. As such, the number of iterators has to be valid. In our sample code, we are passing into DoubleFn() the value as a reference, and doubling it. Lastly, we are calling PrintFn() again, but just for something different, we will start at the

second last one (since we increment the and print the next four. Refer to the following figure:

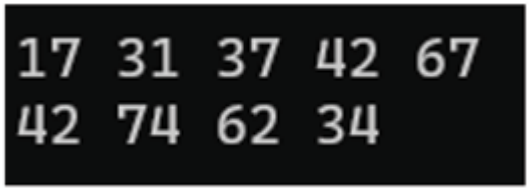

Figure 12.5: Output from DoForEachN

In our output, we start at the second last value. This is the fourth number, and we only modified the first three values. This means that the first number we printed was not modified, but after that, the remaining numbers were doubled.

This is probably not important, but in case you came across the term, or you just wanted something fun to say, many of these functions are called These are described as function-like and inhibit normal argument-dependent lookup, among other qualities. (See references)

Recipe 12.3: Looking at count, any_of, all_of, and none_of functions

Problem: We want an easy way to count in a range.

Solution: There are several ways to do that, depending on your needs.

The next functions to look at are ranges::count() and which are similar to those in the std namespace. The count() function is looking for a particular value, where the count_if() uses a unary predicate (a function that returns a Boolean).

```
{



{

<< << "Examples in: " << __FUNCTION__ <<

const dishes = {

"Linguine with cream

"Macaroni and

"Chef
```

"Chicken with 2 buttermilk

"Tomato

"Broiled

"Escargot topped with garlic butter"

dishLengths;

for d : dishes) {

dishLengths.push_back(d.length());

  }

=

auto longDishCount = dishLengths.end(), lengthToLookFor);

<< "Found " << longDishCount << " dishes that were " << lengthToLookFor
<<

= d) {

  !=

    || !=

```
    || !=

    || != };

=

dishes.begin(), dishes.end(), NonDairyCountFn);

<< "Found " << nonDairyDishCount <<

" dishes that did not contain: cheese, milk, cream, or butter" <<

}

};
```

In our example, the count() function is looking for the value 34 (which in this case corresponds to the length of our longer strings). In the next case, our count_if() function is counting the items that do not contain one of the particular strings we are searching for). The output is:

```
Found 2 dishes that were 34
Found 3 dishes that did not contain: cheese, milk, cream, or butter
```

Figure 12.6: Output for DoCounts

We are going to tweak our lambda and our data for the next example. These too, are similar to those in the std namespace, and they are ranges:: Let us look at the following example:

```
{



{

<< << "Examples in: " << __FUNCTION__ <<

const menu1 = {

"Linguine with cream

"Macaroni and

"Chef

"Chicken with 2 buttermilk

"Tomato

"Broiled

"Escargot topped with garlic butter"

const menu2 = {
```

```
"Linguine with cream

"Macaroni and

"Chicken with 2 buttermilk

"Escargot topped with garlic butter"

const menu3 = {

"Chef

"Tomato

"Broiled chicken"

= d) { return

  !=

    || !=


    || !=

    || != };

= s) { << s << ", };
```

```
for menu : { menu1, menu2, menu3 }) {

= IsDairyFn);

= IsDairyFn);

= IsDairyFn);

<< "For our menu choices:

PrintFn);

<<

<< "Any have dairy is " << << anyOfResult <<

<< "All have dairy is " << << allOfResult <<

<< "None have dairy is " << << noneOfResult <<

<<

  }

}

};
```

We created a few variations of our menu, and for each one, we checked if any/all/none of the menu items had dairy in them. Look at the following output:

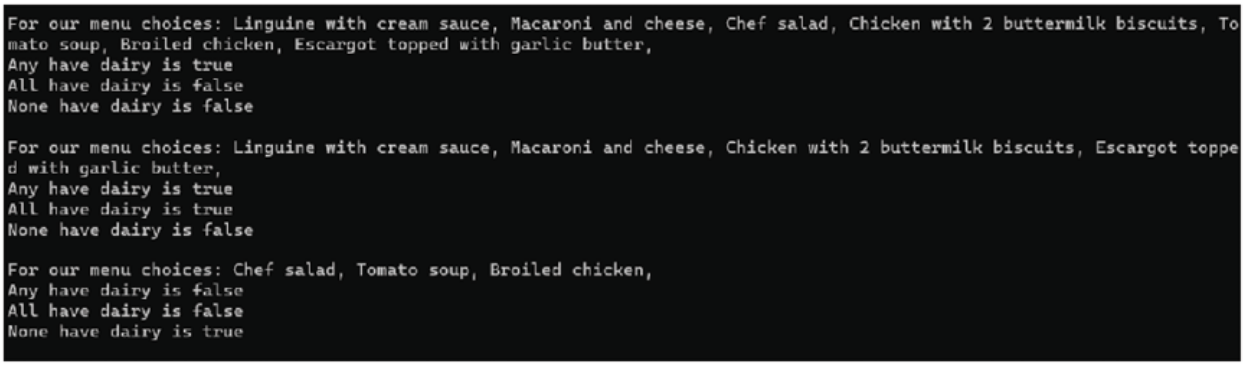
For our menu choices: Linguine with cream sauce, Macaroni and cheese, Chef salad, Chicken with 2 buttermilk biscuits, To
mato soup, Broiled chicken, Escargot topped with garlic butter,
Any have dairy is true
All have dairy is false
None have dairy is false
For our menu choices: Linguine with cream sauce, Macaroni and cheese, Chicken with 2 buttermilk biscuits, Escargot toppe
d with garlic butter,
Any have dairy is true
All have dairy is true
None have dairy is false
For our menu choices: Chef salad, Tomato soup, Broiled chicken,
Any have dairy is false
All have dairy is false
None have dairy is true

Figure 12.7: Output for DoAnyAllNone

We can see that those that have some menu items with dairy in them return ranges::any_of() with true, those that just have dairy menu items return ranges::all_of() as true, and those without dairy menu items return ranges::none_of() as true.

Recipe 12.4: Looking at comparison functions

Problem: We want some flexibility in how we compare with ranges.

Solution: A lot of the range functions are similar to their standard library counterparts.

Our next example shows which compares two ranges and returns an iterator for each where they no longer match.

```
{



{

<< << "Examples in: " << __FUNCTION__ <<

const myQueue1 = { 67 };

const myQueue2 = { 67 };

= n) { << n << ", };

auto result = myQueue2);

<< "Range stopped matching for queue1
```

```
++result.in1, PrintFn);

<<

<< "Range stopped matching for queue2


++result.in2, PrintFn);

<<

}

};
```

If you notice in our output, we are showing the value that did not match. That is because in our print lines, we increment the result.in1 and result.in2 so that it includes the mismatch, as shown:

```
Range stopped matching for queue1 at:17, 31, 37, 42,
Range stopped matching for queue2 at:17, 31, 37, 24,
```

Figure 12.8: Output for LookForMismatch

The next ranges example is also similar to its std counterpart. ranges::equal() takes two parameters and returns a Boolean whether they are equal.

```
{



{

<< << "Examples in: " << __FUNCTION__ <<

const myQueue1 = { 67 };

const myQueue2 = { 67 };

const myQueue3 = { 67 };

const queues = { myQueue1, myQueue2, myQueue3 };



= n) { << n << ", };

for = i <= i++) {

<< "Printing container #" << i + 1 << ":

=

PrintFn);

<<

}
```

```
auto result12 = myQueue2);

auto result23 = myQueue3);

auto result13 = myQueue3);

<< "Queue1 and queue2 are equal is " << << result12 <<

<< "Queue2 and queue3 are equal is " << << result23 <<

<< "Queue1 and queue3 are equal is " << << result13 <<

}

};
```

This gives us the result as shown in Figure

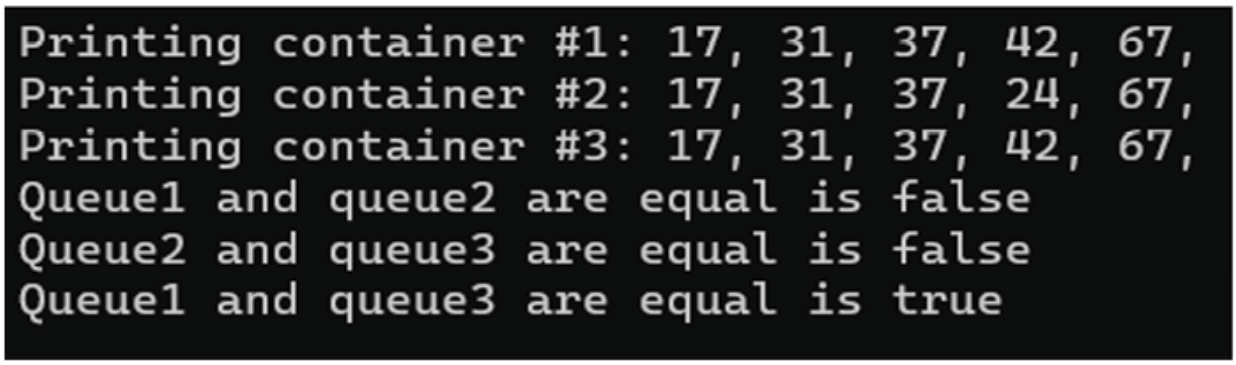
Printing container #1: 17, 31, 37, 42, 67,
Printing container #2: 17, 31, 37, 24, 67,
Printing container #3: 17, 31, 37, 42, 67,
Queue1 and queue2 are equal is false
Queue2 and queue3 are equal is false
Queue1 and queue3 are equal is true

Figure 12.9: Output from LookAtEqual

Continuing with C++20 range functions that have similar standard library equivalents, we can look at which says if the first range is less than the second range, we have:

```
{



{

<< << "Examples in: " << __FUNCTION__ <<

v1 { 'd' };

v2 { 'd' };

v3 {

v4 { 'z' };

vs = { v1, v2, v3, v4 };

= c) { << c << ", };

=

for currentContainer : vs) {

<< "Printing container #" << i++ << ":
```

```
PrintFn);

<<

}

auto result12 = v2);

auto result11 = v1);


auto result13 = v3);

auto result14 = v4);

<< "Container1 < Container2 is " << << result12 <<

<< "Container1 < Container3 is " << << result13 <<

<< "Container1 < Container4 is " << << result14 <<

<< "Container1 < Container1 is " << << result11 <<

}
```

The output is given in the following figure:

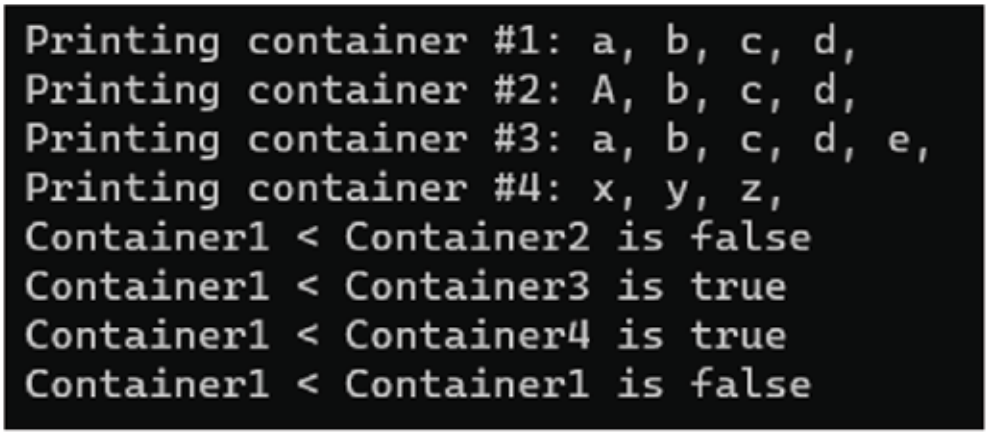
Printing container #1: a, b, c, d,
Printing container #2: A, b, c, d,
Printing container #3: a, b, c, d, e,
Printing container #4: x, y, z,
Container1 < Container2 is false
Container1 < Container3 is true
Container1 < Container4 is true
Container1 < Container1 is false

Figure 12.10: Output from LookAtLexographicalCompare

Looking at the output, we can see that the case does matter, as seen in a and A and note the same. The uppercase letters come before the lowercase letters, and so Container2 would be less than It then compares containers of different sizes, and the one that starts off with the same values is less than the larger container. Obviously a < z so that comparison was easy, but note that if we compare the same container, it returns false, indicating Container1 is not less than

Recipe 12.5: Search functionality

Problem: We need powerful ways to search ranges.

Solution: C++20 and C++23 give you plenty of search functions.

We can look at some C++20 ranges functions that relate to finding a value in our next example:

```
{



{

<< << "Examples in: " << __FUNCTION__ <<

const myQueue = { 88 };

= n) { return n % 7 == };

auto IsMultipleOfSeventeen = n) { return n % 17 == };

= n) { << n << ", };

<< "Full range:
```

```
PrintFn);

<<

bool found37 = != myQueue.end();

<< (found37 ? "Was" : "Was << " able to find "

  << 37 << " in the range" << <<

= myQueue.end() -


auto sub = itAlmostEnd);

<< "Partial range:

PrintFn);

<<

bool found88 = != itAlmostEnd;

<< (found88 ? "Was" : "Was << " able to find "

  << 88 << " in the range" <<

auto foundMultiple = IsMultipleOfSeven);
```

```
if myQueue.end()) {

<< "Found multiple of 7 in the range, "

    << *foundMultiple << ", at index " << (foundMultiple - myQueue.begin())

    <<

  }

else {

<< "Found multiple of 7 was not found in the range" <<

  }

auto foundMultipleInSubRange = IsMultipleOfSeventeen);

if sub.end()) {

<< "Found something that was not a multiple of 17 in the range, "

    << << ", at index "


    << - myQueue.begin())

    <<
```

```
  }

else {

<< "The whole range is multiples of 17" <<

  }

}

};
```

The ranges::find looks for a given value, which our output shows correctly found the first target value. It did not find our second target value, because we provided a smaller range, as shown:

```
Full range: 17, 34, 37, 42, 67, 88,
Was able to find 37 in the range

Partial range: 17, 34, 37,
Was not able to find 88 in the range
Found multiple of 7 in the range, 42, at index 3
Found something that was not a multiple of 17 in the range, 37, at index 2
```

Figure 12.11: Output from LookAtFind

We can see in our output that the find_if() function, that uses a predicate, was able to find the value. We also used the find_if_not() function to find the first range value that failed the predicate.

In C++23, there are now similar functions that start at the last item and iterate backwards. Those are and

The next function is which is given two ranges and tries to find something from the second range within the first range. This could be

A simple value, without a predicate.

A complex value, using a predicate.

Projections of the two ranges, where you select which data member to use.

Our code example shows a few different ways:

```
{



{

<< << "Examples in: " << __FUNCTION__ <<

int x, y; };

const vector1 = { };

const vector2 = { };

const points1 = { };

const points2 = { };
```

```
auto IsMatch = [](Point a, Point b) { return a.x == b.x && a.y == b.y; };

auto PrintNumberFn = n) { << n << ", };

= Point& p)

{ << "(" << p.x << "," << p.y << "), };

<< "Values in vector1:

PrintNumberFn);

<<



<< "Values in vector2:

PrintNumberFn);

<<

auto foundVector12 = vector1.end(),

  vector2.begin(), vector2.end());

if != vector1.end())

  {
```

```
auto distance = foundVector12);

<< "Found value from vector2 (" << *foundVector12

    << ") in vector1 at index " << distance <<

  }

else

  {

<< "Did not find value from vector2 in vector1" <<

  }

<<

<< "Values in points1:

PrintPointFn);

<<

<< "Values in points2:

PrintPointFn);
```

```
<<


auto foundPoint12 = points2, IsMatch);

if != vector1.end())

  {

auto distance = foundPoint12);

<< "Found point from points2 ("

    << foundPoint12->x << "," << foundPoint12->y << ") "

    << "in points1 at index " << distance <<

  }

else

  {

<< "Did not find value from points2 in points1" <<

  }

auto foundProj12 = points2, {},
```

```
  &Point::y);

if != points1.end())

  {

auto distance = foundProj12);

<< "Found y value from points2 in x values of points1 ("

   << foundProj12->x << "," << foundProj12->y << ") "

   << " at index " << distance

   <<


  }

else

  {

<< "Did not find value from points1 in points1" <<

  }

}
```

};

This results in the following output:

```
Values in vector1: 2, 3, 5, 7, 1, 8,
Values in vector2: -2, 3, 4, 5, 7,
Found value from vector2 (3) in vector1 at index 1

Values in points1: (2,3), (5,7), (1,8),
Values in points2: (1,3), (4,4), (5,7), (8,1),
Found point from points2 (5,7) in points1 at index 1
Found y value from points2 in x values of points1 (1,8)  at index 2
```

Figure 12.12: Output from LookAtFindFirstOf

The first call to ranges::find_first_of() is pretty simple, but the second call uses a predicate to match the x and y values. The last call to ranges::find_first_of() provides an empty predicate, and finds the first y value (from points2: in the projected range of x values has

There is also C++20's which looks for the last occurrence from the second range [first2, last2) in the first range [first1, It too, has the overloads where it can accept a predicate and/or projections.

Whereas ranges::find_first_of() looks for the first element from range2 in the range1 values, C++20 also provides This looks for the first occurrence of the sequence of all the elements in range2 in the range1 values.

{

{

```
<< << "Examples in: " << __FUNCTION__ <<

const vector1{ 6 };

const term{ 3 };

= n) { << n << ", };

auto IsAbsEqualFn = a, b) { == };

<< "Values in vector1:

PrintFn);

<<

<< "Values in search term:

PrintFn);

<< <<

auto found1 = term);

if found1.empty()) {

foundStart = found1.begin());
```

```
foundEnd = found1.end());

<< "Found search term in vector1, starting at index ["

    << foundStart << "] (" << *(found1.begin()) << ") to index ["


    << foundEnd - 1 << "] (" << - << ")" <<

  }

else

  {

<< "Could not find term in vector1" <<

  }

auto found2 = term, IsAbsEqualFn);

if found2.empty()) {

foundStart = found2.begin());

foundEnd = found2.end());

<< "Found search term (via predicate) in vector1, starting at index ["
```

```
    << foundStart << "] (" << *(found2.begin()) << ") to index ["
    << foundEnd - 1 << "] (" << - << ")" <<
  }
else
  {
<< "Could not find term (via predicate) in vector1" <<
  }
}
};
```

In the following output, we can see that it finds the sequence starting at index 4, but when we use our predicate that compares the absolute values, it says that it found the sequence starting at index 1, as shown:

```
Values in vector1: 3, -1, -2, -3, 1, 2, 3, 4, 5, 1, 2, 3, 6,
Values in search term: 1, 2, 3,

Found search term in vector1, starting at index [4] (1) to index [6] (3)
Found search term (via predicate) in vector1, starting at index [1] (-1) to index [3] (-3)
```

Figure 12.13: Output from LookAtSearch

Somewhere in-between the ranges::find_... variants and ranges::search() is This is looking for a single value to appear in the range multiple times. You can have the same result if you create a range to search for all that had repeated values, but this saves a step. As with the other functions in this category, it too has overloads that take a predicate and/or a projection.

The next function to look at is C++20's which is similar to C++17's This iterates the range, comparing the current value to the value next to it. It can either match by exact value, without a predicate or if the predicate returns true. There is also an overload that takes projections as well.

```
{



{

<< << "Examples in: " << __FUNCTION__ <<

const vector1 = { 8 };



= n) { << n << ", };

auto IsAbsEqualFn = a, b) { == };

<< "Values in vector1:

PrintFn);

<<
```

```
auto found1 =

if != vector1.end()) {

auto distance = found1);

<< "Found equal values in vector1 (" << *found1

    << ") at index " << distance <<

  }

else {

<< "Did not find equal adjecent values in vector1" <<

  }

<<

auto foundAbs = IsAbsEqualFn);

if vector1.end()) {

auto distance = foundAbs);

<< "Found absolute-equal values in vector1 (" << *foundAbs
```

```
    << ") at index " << distance <<
  }

else {
<< "Did not find absolute-equal adjecent values in vector1" <<
  }
}
};
```

In our first example, we just do a value match. In the second example, we provide a predicate that compares absolute values. This is why it finds a match between 5 and -5 before it matches on the two number 8 values on the end, as shown in the following output:

```
Values in vector1: 2, 3, 5, -5, 7, 1, 8, 8,
Found equal values in vector1 (8) at index 6

Found absolute-equal values in vector1 (5) at index 2
```

Figure 12.14: Output of LookAtAdjacentFind

While we have ranges::find() that returns an iterator to the found position (or end() if not found), developers can rejoice with C++23's That has similar

functionality but returns a Boolean, which is less typing to check if you succeeded in your search.

```
{



{

<< << "Examples in: " << __FUNCTION__ <<

const myQueue = { 88 };



= n) { };

= n) { << n << ", };

myQueue2 = { -42 };

<< "Full range:

PrintFn);

<<

bool found37 =

<< (found37 ? "Was" : "Was << " able to find "
```

```
  << 37 << " in the range" << <<

= myQueue.end() -

auto sub = itAlmostEnd);

<< "Partial range:

PrintFn);

<<

bool found88 =

<< (found88 ? "Was" : "Was << " able to find "

  << 88 << " in the range" <<

<< "Small range:

PrintFn);

<<

bool foundsub = myQueue2, {}, IsAbsEqualFn, IsAbsEqualFn);



<< (foundsub ? "Was" : "Was << " able to find "
```

```
  << " the small range in the main range (via the projections)" <<

}

};
```

Our LookAtContains() example has similar searches as the LookoAtFinds() code, and the results are as follows:

```
Full range: 17, -34, 37, 42, 67, 88,
Was able to find 37 in the range

Partial range: 17, -34, 37,
Was not able to find 88 in the range
Small range: 34, -37, -42,
Was able to find  the small range in the main range (via the projections)
```

Figure 12.15: Output from LookAtContains

Our last pair of functions that we will look at is also a welcome new addition in C++23. The ranges::starts_with() looks for a range at the beginning of another range, while ranges::ends_with() returns a Boolean if it finds a range at the end of another range.

```
{



{

<< << "Examples in: " << __FUNCTION__ <<

const myQueue = { 88 };
```

```
= n) { };


auto IsAbsMatch = a, b) { == };

= n) { << n << ", };

myQueueStart = { -37 };

myQueueEnd = { 88 };

<< "Full range:

PrintFn);

<<

<< "Start search range:

PrintFn);

<<

bool foundStart = myQueueStart);

bool foundStartAbs = myQueueStart, {}, IsAbsEqualFn, IsAbsEqualFn);

<< (foundStart ? "Was" : "Was << " able to find the search range"
```

```
" at the start of the main range" <<

<< (foundStartAbs ? "Was" : "Was << " able to find the search range"

" at the start of the main range, using absolute comparison" <<

<< << "End search range:

PrintFn);

<<

= myQueueEnd);


bool foundEndAbs = myQueueEnd, IsAbsMatch);

<< (foundEnd ? "Was" : "Was << " able to find the search range"

" at the end of the main range" <<

<< (foundEndAbs ? "Was" : "Was << " able to find the search range"

" at the end of the main range, using absolute comparison" <<

}

};
```

The output shows:

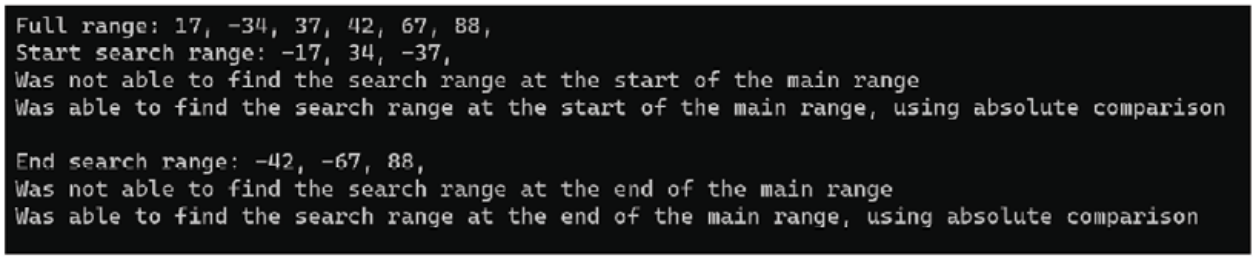
Full range: 17, -34, 37, 42, 67, 88,
Start search range: -17, 34, -37,
Was not able to find the search range at the start of the main range
Was able to find the search range at the start of the main range, using absolute comparison
End search range: -42, -67, 88,
Was not able to find the search range at the end of the main range
Was able to find the search range at the end of the main range, using absolute comparison

Figure 12.16: Output from LookAtStartsEndsWith

The ranges::starts_with() fails when it does a strict comparison, but does succeed when we use a projection predicate on both values resulting in an absolute value comparison.

Similarly the ranges::ends_with() fails for the first attempt, but then when we call a predicate that does an absolute comparison it has success.

Understandably, with that many functions, it can get confusing on which one to use, but we also included the following Table 12.1 for quick lookup:

lookup: lookup: lookup:
lookup:
lookup:

lookup:
lookup:
lookup:

lookup:
lookup:
lookup:
lookup:
lookup:
lookup:
lookup:
lookup:
lookup:

Table 12.1: Summary of search functions

Note: Some might say that a lot of these functions are not needed. With all the shortcomings of the standard library over the years, people have written their utility functions for some of these features. The newer code (probably/hopefully) takes advantage of other newer features; such as const-related keywords, possibly threading, and other newer compiler advantages. The library code, in all likelihood, will be faster than yours, safer, and certainly tested and used more. We recommend using the standard library code instead of owning and maintaining your code. Lastly, all of these ideas about concepts, ranges, coroutines, and many others, will be a big part of the language for years to come. It is beneficial to embrace them and see how you can make your code better.

Conclusion

At the beginning of the chapter, we stated the goal that you would know more about ranges. We went through a lot of functions and code samples, and that was only a fraction of what ranges has to offer! We saw that many of the range functions, started with standard or container functions, and are now using concepts that can work with containers, as well as ranges. We started talking about just how to iterate over a range, how to get the size or count something, and tell if it is empty. We also learned how to get at the raw data. Next, we dove into all of the ways to looking for something in a range. Many of those come from the standard library, so that you might be able to use something similar. However, the range functions offer lots of flexibility.

It is good to see that just about all range functions that could use a predicate or a projection, has been included in Table That will cut down on a lot of other custom coding, when you can create little helper methods, or use a built-in helper.

Points to remember

Ranges were a big C++20 feature, with some extra functionality added in the C++23.

It is good to learn what the various search functions do, and use the one that is most appropriate. For example, if you are just looking to see if something exists in a range, then you do not have to get a count. You can use contains or which should stop after the first match. Steps like that speed up your code, and make it cleaner.

When searching, see if your scenario can benefit from a projection and/or a predicate. If it helps identify other cases, that might mean it reduces other code, or it might help make your code more solid.

References

CPPreference. (n.d.). Niebloid. Retrieved from https://en.cppreference.com/w/cpp/algorithm/ranges/for_each

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 13

Range Algorithms: Sort, Search and More

Introduction

As we have learned so far, there is no shortage of range functions. So far, we have learned how to make them, how to iterate them, how to count them, and ways to search in them. In this chapter, we will continue looking at functions in the std::ranges library.

Structure

In this chapter, we will cover the following topics:

Recipe 13.1: Minimum and maximum functions

Recipe 13.2: Sorting and partitioning functions

Recipe 13.3: Binary search functions, and set functions

Recipe 13.4: Permutation functions

Recipe 13.5: Fold functions

As you can see, we are fortunate that all the features we will be talking about in this chapter work with the major compilers.

```
{

// works with          when tested

    //2023.12.09 yes

    //2023.12.09 yes
```

//2023.12.09 yes

//2023.12.09 yes

yes

//2023.12.09 yes

//2023.12.09 yes

//2023.12.09 yes

}

Objectives

By the end of this chapter, the reader will know even more about the power of ranges and the functionality they provide. You will learn about some of the basic functions, relating to searching and sorting. You will also see some complex functions, such as folding and dealing with permutations. You may not use these every day, but you will be glad that you can leverage them and not have to worry about creating them yourself.

Recipe 13.1: Minimum and maximum functions

Problem: We want an easy way of getting the minimum and maximum value from a range.

Solution: The C++20 release has range functions to do that.

Continuing our tour of the ranges library, we will start with a couple more functions that were taken from the standard library but made for ranges. You have the choice of:

getting the minimum or maximum value.

getting the iterator that points at the minimum or maximum value.

getting the minimum and maximum values with the same call.

getting the iterators that point at the minimum and maximum value.

Before we get to the code snippet, let us look at the headers:

```
#include

#include

#include
```

```
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include
```

#include

The code implementation is as follows:

{

{

<< << << __FUNCTION__ <<

myQueue = { 99 };

= n) { << n << ", };

<< "Full range:

PrintFn);

<<

=

=

<< "Min=" << minNum << ", and Max=" << maxNum <<

=

```
=

= minIt);


= maxIt);

<< "Found Min at index [" << minDistance << "]= " << *minIt

  << ", and Max at index [" << maxDistance << "]= « << *maxIt <<

<< << "Or if want to figure out both at same time:" <<

minMax =


<< "Min=" << minMax.min << ", and Max=" << minMax.max <<

auto [min2It, max2It] =

min2Distance = min2It);

max2Distance = max2It);

<< "Found Min at index [" << min2Distance << "]= " << *min2It

  << ", and Max at index [" << max2Distance << "]= « << *max2It << <<
```

```
myVector = {};

// cannot get min/max on empty range, or else exception

// auto maxVector = std::ranges::max(myVector);

=



= myVector.begin();

= myVector.end();

if {

if == vEnd && maxItVector == vBegin) {

<< "Makes sense for the max to point to the beginning or end"

    << << " (both null) since no items in the range." <<

}

  }

<< << "Clamping results:" <<

=
```

=

for {

auto clampedResult = lowerClamp, higherClamp);

if == n) {

<< "Value of " << n << " was within range of "

<< lowerClamp << "-" << higherClamp <<

}

== lowerClamp) {

<< "Value of " << n << " was below lower limit " << lowerClamp <<

}

== higherClamp) {

<< "Value of " << n << " was above upper limit " << higherClamp <<

}

}

```
}

};
```

We have included the same headers as the last chapter because we will probably use most of them again. The first code block simply grabs the min and max values separately. The next block finds the iterators that point at the min and max, and we also use the std::distance() function to get the index. Note that although we have the value 99 twice in the range, max_element() returns the earliest/leftmost iteration point.

The third block calls that returns which we can get the min and max with one call. We also called std::ranges::minmax_element() but instead of declaring a ranges::minmax_element_result variable, we use a structured binding. Curiously, we got a different index for our maximum value this time. Just note that the specification for this function returns the rightmost/latest iteration point for the maximum. The output from LookAtMinMax is shown in the following Figure

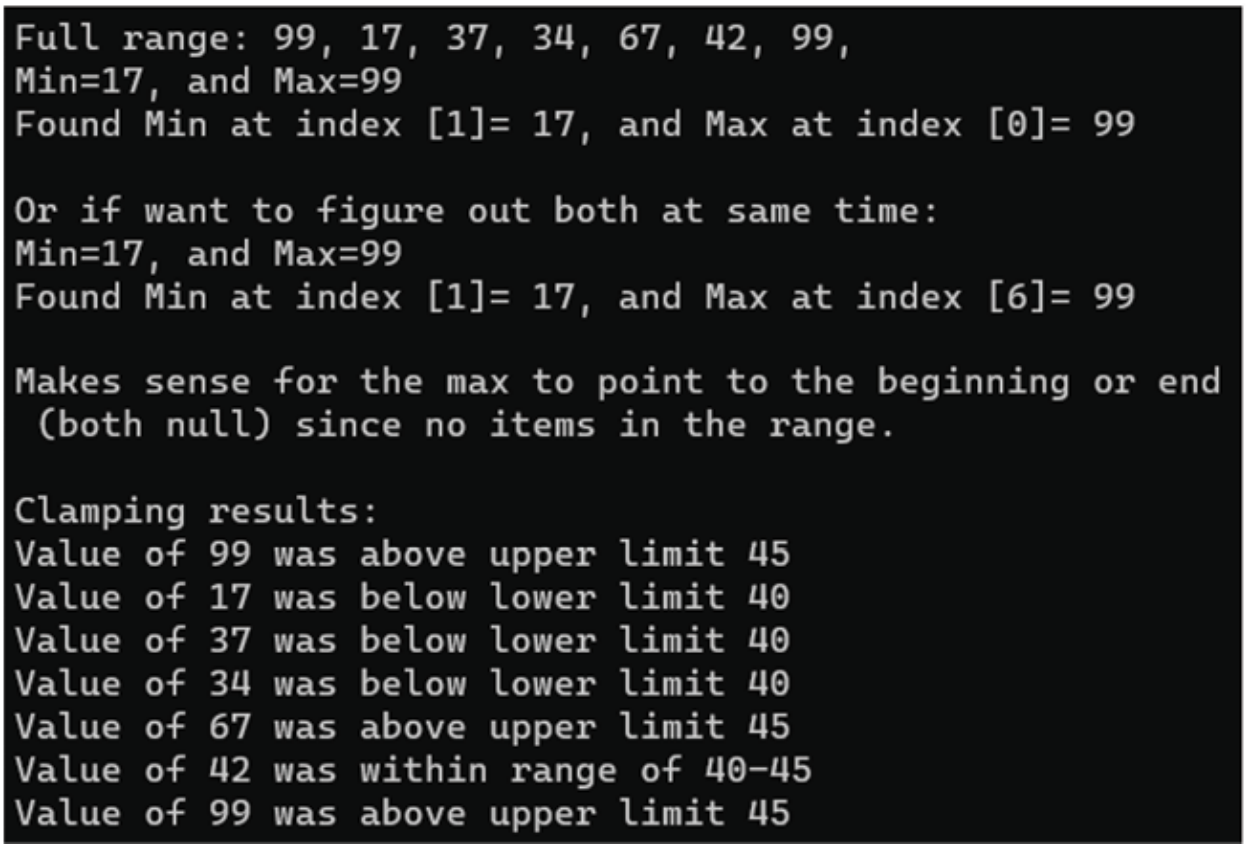
Full range: 99, 17, 37, 34, 67, 42, 99,
Min=17, and Max=99
Found Min at index [1]= 17, and Max at index [0]= 99
Or if want to figure out both at same time:
Min=17, and Max=99
Found Min at index [1]= 17, and Max at index [6]= 99
Makes sense for the max to point to the beginning or end
(both null) since no items in the range.
Clamping results:
Value of 99 was above upper limit 45
Value of 17 was below lower limit 40
Value of 37 was below lower limit 40
Value of 34 was below lower limit 40
Value of 67 was above upper limit 45
Value of 42 was within range of 40-45
Value of 99 was above upper limit 45

Figure 13.1: Output from LookAtMinMax

Be careful that if the range is empty, it makes sense that the min and max iterators would be pointing to the end (or just null for this case). Moreover, if you ask for the min/max/minmax value, then it will throw an exception.

All of these functions in our example also have overloads that you can provide comparison for and projection predicates. This means that although the basic functionality does not differ much, compared to their standard library counterparts, having the overloads make them very powerful and could come in handy in different ways.

A case of this would be The clamp function is similar to where it is given a value, as well as lower and upper limits, which returns the lower limit value,

the value itself, or the upper limit value, depending on whether the given value sits. Although our example did not use any projection, having that ability makes the ranges version more powerful than its predecessor.

Recipe 13.2: Sorting and partitioning functions

Problem: We need different ways to sort and group our ranges.

Solution: C++20 has several ways to order your data.

Some of the range functions will be familiar to you, and if not, then they should be fairly easy to understand. Moreover, you can assume these functions will have overloads for a custom comparer, and the ability for projections.

Before we get into the sorting code, we will create a PrintRange() function, just so we do not have to repeat it several times in our example. The only big thing to note is that we check the value type used in the range; if it is an integer, then we will print the numbers separated with a comma. If not, then we will just have a space separating the items, as shown:

```
{



auto r, title = {

using R =

using Elem =

", " };
```

```
if {

sep = ",

  }

else {

sep = "

  }

= x) { << x << sep; };

if {


<< title << "

  }

else {

<< "Full range :

  }

PrintFn);
```

<<

}

};

We can start with a very simple number sort of integer number. We print the range before and after the sort, to show the result. We also added to the output the result of If you think your sort operation may take longer than desired, you might want to check if your range is already sorted. The ranges::is_sorted() can use the same sort and projection parameters as the sort, but will just return saying whether it is already sorted.

In the following code, we also perform a sort on a range of strings. The next thing we do is create a container of our structures that holds a number and a string. We then sort the structure, using a projection. Just in case you were wondering, in our NumberAndWord struct, we created an operator<< function to act as a little helper to output the struct members. (See references)

{

auto r)

{

if {

```
<< "Range is sorted in descending order" <<

  }

{

<< "Range is sorted in ascending order" <<

  }

else {

<< "Range is not sorted" <<

  }

}



{


word;

friend

(Os& os, NumberAndWord const& p) {

  return os << std::left << std::setw(8) << p.number << « : «
```

```
    << p.word << ‹\n›;

  }

};

static void LookAtSort() {

std::cout << std::endl << std::endl << __FUNCTION__ << std::endl;


std::deque myQueue = { 99, 17, 37, 34, 67, 42, 99 };



CheckIfSorted(myQueue);

std::ranges::sort(myQueue);

  "Afer sort: «);

CheckIfSorted(myQueue);

std::cout <<

std::vector words = { "C++", "23", "is", "a", "fantastic", "release", "!" };

  "My sentence before sort: ");
```

```
std::ranges::sort(words);

  "Afer doing a word sort : ");

std::cout <<

myQueue2 = { 99, 17, 37, 34, 67, 42, 99 };

std::vector myPairs;

  for (int i = 0; i < myQueue2.size(); i++) {

myPairs.push_back({ myQueue2[i], words[i] });

  }

  "Before a projected sort: \n");

std::ranges::sort(myPairs, std::ranges::greater(), &NumberAndWord::word);

  "Afer a projected sort: \n");

std::cout <<


}

};
```

The output is as follows:

```
Full range : 99, 17, 37, 34, 67, 42, 99,
Range is not sorted
Afer sort:  17, 34, 37, 42, 67, 99, 99,
Range is sorted in ascending order

My sentence before sort:  C++ 23 is a fantastic release !
Afer doing a word sort :  ! 23 C++ a fantastic is release

Before a projected sort:
 99       : !
 17       : 23
 37       : C++
 34       : a
 67       : fantastic
 42       : is
 99       : release

Afer a projected sort:
 99       : release
 42       : is
 67       : fantastic
 34       : a
 37       : C++
 17       : 23
 99       : !
```

Figure 13.2: Output for LookAtSort

You can see in our sorted output of strings, that it goes from symbol, to numbers, to uppercase letters, to lowercase letters.

Our projected sort was done in ascending order, using the ranges::greater() function as a sort specifier for the name member variable that we projected. The output shows the words in reverse order as before, but the number variable still matches with the word it was paired with.

One other function to mention is which performs similarly to The one difference is that with it keeps equal elements in the same sequence. A quick example of this is we have a structure Name and Age elements, and in the range we had: …, { 25, "Zach"}, … {25, If we used ranges::sort() by then the result will have Alex and Zach together, but there is no guarantee who will be first. With despite having the same age, and Alex coming before Zach (which does not factor into it, since we are only sorting by then Zach will still come before but that is how the order originally was. It probably will not be something you will use every day, but at least it is there if it helps.

Continuing with the sort functions, we split up the code to make it more readable on a page, as follows:

```
{



{

<< << << __FUNCTION__ <<

myQueue3 = { 99 };

  "Before partial sort of 3 items:

myQueue3.begin() +

  "After partial sort of 3 items:
```

```
auto partiallySortedIt =



auto partiallySortedDistance = partiallySortedIt) -

<< "Seems to be sorted for: " << partiallySortedDistance << " items." <<

<<

myQueue4 = { 99 };

  "Source pre partial_sort_copy :

myVectorDest = { 3 };

  "Dest before partial_sort_copy:

auto partialSortCopyResult = myVectorDest);

  "Source after partial_sort_copy  :

  "Dest after the partial_sort_copy:

<<

myQueue5 = { 99 };

  "Before nth sort:
```

```
auto partialNthSortResult = myQueue5.begin() +


  "After nth sort :

<<

myQueue6 = { 99 };

  "Before partitioned:

auto Partition42Fn = n) { return n <= };

auto partitionSub = Partition42Fn);

  "After partitioned :

auto isPartitioned = Partition42Fn);

<< "Size of partition tail =" << partitionSub.size()

  << ", and " << : "is << " partitioned" <<

auto partitionPoint = Partition42Fn);

auto partitionDistance = partitionPoint);

<< "Partition is split after " << partitionDistance << " items" <<
```

```
}

};
```

This example first performs where it will sort a range until we hit the destination iterator. A similar result would occur if you sorted a sub-range. We also called the ranges::is_sorted_until() function, which tells you the iterator where the provided range is sorted until. We subtracted one from the distance to show the index is the same number that we provided to the

In the next code block, we called That copies the values into the destination range, leaving the source range unchanged. Naturally, it copies into the destination the number of values that the destination can hold.

The next code block is a specialized partial-partial sort named ranges:: This ensures that just the Nth item in the range is the correct value, as if the whole range was sorted, and it specifies the values before the Nth item are all less than the Nth item. However, it does not guarantee that the first (N-1)th items are sorted, just that they are before the Nth. In the output that you see in Figure when we asked it to sort to get the item, since our range is small, the other items appear to be sorted.

Similar to the ranges::nth_element() function, we will now look at partitions. These do not have anything to do with module partitions. Given a predicate, it will separate the range into two partitions. Our example code partitions numbers that are <= 42, but you can partition odd versus even, or using projections on a member variable you provide. Instead of returning an iterator, like ranges::nth_element() does, it returns the second sub-range, after the partitioning split. It has no guarantee of sorting, just grouping those that meet the predicate's criteria, and those that do not.

Refer to Figure

```
Before partial sort of 3 items:  99, 17, 37, 34, 67, 42, 99,
After partial sort of 3 items:  17, 34, 37, 99, 67, 42, 99,
Seems to be sorted for: 3 items.

Source pre partial_sort_copy :  99, 17, 37, 34, 67, 42, 99,
Dest before partial_sort_copy:  1, 2, 3,
Source after partial_sort_copy  :  99, 17, 37, 34, 67, 42, 99,
Dest after the partial_sort_copy:  17, 34, 37,

Before nth sort:  37, 99, 17, 34, 67, 42, 99,
After nth sort :  17, 34, 37, 42, 67, 99, 99,

Before partitioned:  37, 99, 17, 34, 67, 42, 99,
After partitioned :  37, 42, 17, 34, 67, 99, 99,
Size of partition tail =3, and is partitioned
Partition is split after 4 items
```

Figure 13.3: Output of LookAtPartialSort

Our output first shows the partial sort after 3 items, and retrying the algorithm with it confirms the first 3 items are sorted.

The next block shows where it filled the destination range with the first few values from the source range.

We told the ranges::nth_element() function to ensure the element was sorted.

And lastly, on calling ranges::partition() asking to group all values <= 42, it says the split was after 4 items, meaning the sub-range it returned from the call has 3 items in it.

Some of these are specialized functions, but their goal is not to do a complete sort on all the items, but put the items into buckets. That could be very handy where, instead of performing a sort on a large collection, you can just make a partition into something manageable. The items may not be sorted, but you would have quickly narrowed down what you are looking for. It may not have the same power as SQL's Partition, or Group By commands, or a C# group-by call using LINQ, since our C++ command only splits into two groups, but that does not stop you from splitting that up into further sub-ranges.

In this section, there is also ranges::partition_copy() which, similar to will copy values, based on the predicate, into the target range. There is also which still partitions based on the predicate, but the stable aspect means that the order of items is preserved.

Recipe 13.3: Binary search functions, and set functions

Problem: It would be good to have search-related functions, once we had our data sorted.

Solution: Akin to some of the functions we looked at in the last topic, C++20 had several ways to investigate your sorted data.

Our previous topic showed how you can sort, or at least partition your data. Now we will look at some C++20 range functions that allow you to use that sorted data. If you look at our example code:

```
{



{

<< << << __FUNCTION__ <<

myQueue7 = { 27 };


  "After sorted:

if {
```

```
<< "Found 42 via binary search" <<

<<

  }

lower =

upper =


  upper), "Lower to Upper bounds:

  lower), "Start to Lower bounds:

  myQueue7.end()), "Above upper bounds:

<<

equalRangeResult =

beforeEqualRange = equalRangeResult.begin());

AfterEqualRange = + myQueue7.end());

  "EqualRange:

  "Before EqualRange:
```

```
  "After EqualRange:

<<

myMiniQueue1 = { 37 };



includeResult1 = myMiniQueue1);

<< "The big range " << (includeResult1 ? "includes" : "does not << " the

  "smaller Range:

myMiniQueue2 = { 99 };

includeResult2 = myMiniQueue2);

<< "The big range " << (includeResult2 ? "includes" : "does not << " this

  "other mini Range:

}

};
```

The functions we will be looking at in this section need the data to be sorted. After we do that, we can try the ranges::binary_search() function. This will be handy at times to get a quick answer on whether a value is in the range. It is also handy that we are just looking at a range, whose underlying content may

not have been easy to perform a binary search on. The next few functions perform similar operations. If we provide a value to ranges::lower_bound() or we will get an iterator before the value (for lower) or after the value (for upper).

In our output shown in Figure we also printed the sub-range of the lower to upper bound, which includes our target value. We then use which gives us similar results, but with one function call that returns a subrange, instead of two calls that give us two iterators that we can use to make a range. The one minor difference is, that with the begin() iterator points at the value, and so we just incremented over that so the two output parts look similar.

```
After sorted:  4, 17, 23, 27, 34, 37, 42, 67, 99, 99,
Found 42 via binary search

Lower to Upper bounds:  42,
Start to Lower bounds:  4, 17, 23, 27, 34, 37,
Above upper bounds:  67, 99, 99,

EqualRange:  42,
Before EqualRange:  4, 17, 23, 27, 34, 37,
After EqualRange:  67, 99, 99,

The big range includes the smaller Range:  17, 37,
The big range does not include this other mini Range:  1, 99,
```

Figure 13.4: Output for LookAtRangeSearch

The final bit of output you can see is from calling This tells us if one range is included in another. You can see by the first call that the values do not have to be contiguous. Rather, it checks each value individually.

Our next code sample looks at set operations. The input of these functions is two ranges, and the output is another range:

```
{

{

<< << << __FUNCTION__ <<

myQueue8 = { 27 };

  "Sorted First Range:

5> myList9 = { 65 };

  "Sorted Second Range:

<<

myVectorIntersection{};

myList9,

  "Set intersection:

myVectorUnion{};
```

```
myList9,

  "Set union:

myVectorDiff{};

myList9,

  "Set difference {first (not in) second}:


myVectorDiff2{};

myQueue8,

  "Set difference {second (not in) first}:

myVectorSymmetricDiff{};

myList9,

  "Set symmetric difference (first,second):

}

};
```

First, we look at the the values that appear in both ranges. Note that the value of 99 is only included once since although it is in the first range twice, it is only in the second range a single time. Next, we look at the the values from both ranges are combined, but not duplicated. Here, we have the value 99 twice, since it is only one occurrence in the second range. We then call ranges::set_difference() a couple of times, first to find the values in the first range that are not found in the second, and then to find the values of the second range not found in the first. Lastly, we call which shows us any value from either that is not in both. This is the equivalent of ranges::set_union() -

Using the ranges library, make these operations simple, considering we are comparing values between a std::deque and a with our results going into a

```
Sorted First Range:  4, 17, 18, 23, 27, 34, 37, 42, 67, 99, 99,
Sorted Second Range:  1, 42, 65, 67, 99,

Set intersection:  42, 67, 99,
Set union:  1, 4, 17, 18, 23, 27, 34, 37, 42, 65, 67, 99, 99,
Set difference {first (not in) second}:  4, 17, 18, 23, 27, 34, 37, 99,
Set difference {second (not in) first}:  1, 65,
Set symmetric difference (first,second):  1, 4, 17, 18, 23, 27, 34, 37, 65, 99,
```

Figure 13.5: Output from LookAtSetOperations

We will look at one last code example for this topic, which relates to merge functions for ranges. The non-inline merge is fast, and we did the same merge twice to show you options. The simpler case uses the std::back_inserter() function, but if you think that will have too much of a performance hit, then the alternative is to resize it beforehand, on the possibility it will store the complete combination of both. Refer to the following code:

{

```
{

<< << << __FUNCTION__ <<

myQueue10 = { 67 };


  "Sorted First Range:

6> myList11 = { 27 };



  "Sorted Second Range:

   {

myVectorMerge{};

ret = myList11,

  "Merged output (presized)  :

   }

{
```

```
myVectorMerge{};

myVectorMerge.resize(myQueue10.size() + myList11.size());

ret = myList11, myVectorMerge.begin());

  ret.out), "Merged output (back-insert):

  }

<<

myVector12 = {   65 };

  "Before inline merge, sorted by

auto inlineMergeResult = myVector12.begin() + myVector12.end());

  "After we executed the inline merge



}

};
```

The other function shown is This might be more of a specialized function, where you have partitioned your data. The data sits in one container, but you have two halves that are individually sorted. Our vector shows we have values 1-99 in one half, and then values 23-65 in the second half. In the

following output, you will see that after the merge, they are correctly sequenced:

```
Sorted First Range:  17, 34, 37, 67, 99,
Sorted Second Range:  4, 18, 23, 27, 42, 99,
Merged output (presized)   :  4, 17, 18, 23, 27, 34, 37, 42, 67, 99, 99,
Merged output (back-insert):  4, 17, 18, 23, 27, 34, 37, 42, 67, 99, 99,

Before inline merge, sorted by halves: 1, 4, 17, 18, 67, 99, 99, 23, 27, 34, 37, 42, 65,
After we executed the inline merge   : 1, 4, 17, 18, 23, 27, 34, 37, 42, 65, 67, 99, 99,
```

Figure 13.6: Output from LookAtMerge

Recipe 13.4: Permutation functions

Problem: It would be useful to have a way to generate and test for permutations.

Solution: C++ 20 created functions to work with ranges and do that.

We have already seen that we can use ranges to generate sets with ranges, in the form of unions and intersections, among others. We will show you that ranges can also work with permutations, where it can show all the different ways the values of a range can be ordered. We need to first sort our range, in normal/descending order if we use and in ascending order if we use If the range is not sorted, it will not calculate all the possibilities. If you look at the code example, we sort the range, and then just loop until it says the next permutation is not found.

```
{



{

<< << << __FUNCTION__ <<

myQueue13 = { 67 };

  "Original
```

```
<<

int counter =
do {
ss;
  ss << "Next

  ss <<
  ss << ":
  ss.str());
  } while
<<

  counter =
do {
```

```
ss;

 ss << "Prev

 ss <<

 ss << ":

 ss.str());

 } while

4> myArray14 = { 67 };

3> myArray15 = { 67 };

myVector16 = { 67 };

myArray14PermResult = myArray14);

myArray15PermResult = myArray15);


myVector16PermResult = myVector16);

<< "This " << (myArray14PermResult ? "  « : "is << " a
```

```
  "

//

<< "This " << (myArray15PermResult ? "  « : "is << " a

  "

//

<< "This " << (myVector16PermResult ? "  « : "is << " a

  "

}

};
```

Looking at our output in Figure you can see that with our range of 4 numbers, we have 24 (0 to 23) rows of permutations. This makes sense because 4! (4 factorial) equals 24 (4*3*2*1).

Next in our code, we create a few different ranges of values. We check each one to see if the given range is a permutation of the range we were originally working with. Notice that it does not have to be sorted to try this, since it needs to go through all the permutations anyway. The original

range that we are comparing to is no longer sorted either, but our first attempt does say that it is a valid permutation. Our second attempt, using the values { 42, 99, 67 } is a subset of our range, but since it does not have the same number of items, and naturally would not be a permutation. Our last attempt, with the values { 18, 42, 66, 67 } does have the right number of items, but our original range does not have the value 66. Thus, it correctly says in the output shown in Figure that it is not a valid permutation.

Just for reference, in the C++20 release, there are also std::prev_permutation() and std::is_permutation() functions. Often, when talking about range functions, we have said that there was already a similar function in the standard library. People must have thought these functions would be very popular since this is a rare case where they were added to the ranges library and the standard library at the same time.

```
Original range: 99, 18, 42, 67,

Next #0:  18, 42, 67, 99,
Next #1:  18, 42, 99, 67,
Next #2:  18, 67, 42, 99,
Next #3:  18, 67, 99, 42,
Next #4:  18, 99, 42, 67,
Next #5:  18, 99, 67, 42,
Next #6:  42, 18, 67, 99,
Next #7:  42, 18, 99, 67,
Next #8:  42, 67, 18, 99,
Next #9:  42, 67, 99, 18,
Next #10:  42, 99, 18, 67,
Next #11:  42, 99, 67, 18,
Next #12:  67, 18, 42, 99,
Next #13:  67, 18, 99, 42,
Next #14:  67, 42, 18, 99,
Next #15:  67, 42, 99, 18,
Next #16:  67, 99, 18, 42,
Next #17:  67, 99, 42, 18,
Next #18:  99, 18, 42, 67,
Next #19:  99, 18, 67, 42,
Next #20:  99, 42, 18, 67,
Next #21:  99, 42, 67, 18,
Next #22:  99, 67, 18, 42,
Next #23:  99, 67, 42, 18,
```

Figure 13.7: Output from LookAtPermutations

Refer to Figure

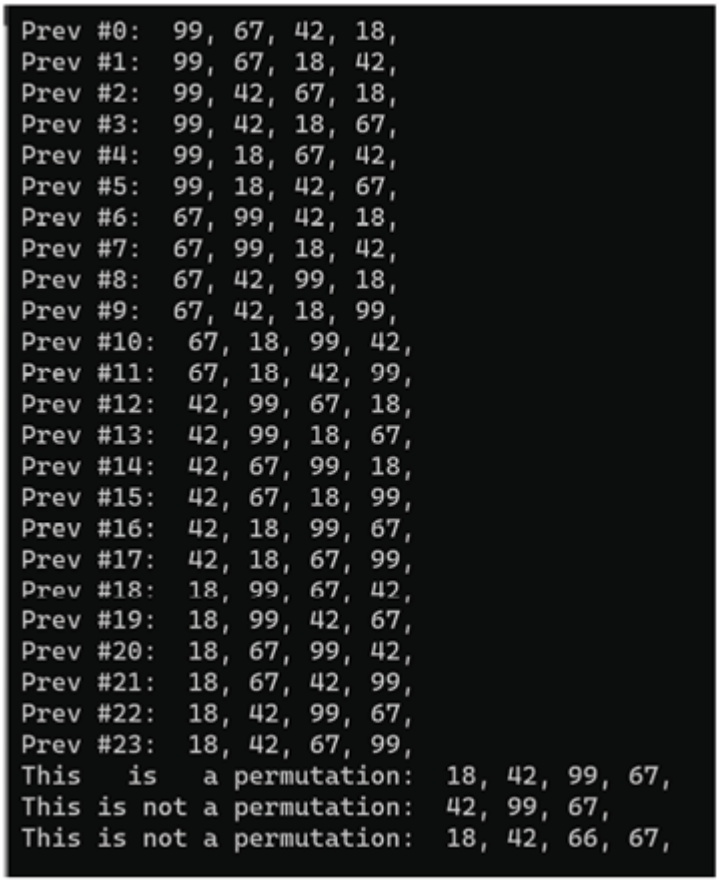

Figure 13.8: Second half of output from LookAtPermutations

Recipe 13.5: Fold functions

Problem: We would like range functionality relating to folding elements.

Solution: While not in the original ranges release, C++ 23 added a number of fold operations for ranges.

Folding means to move from one side of the range to the other, performing the same function with the aggregate total. This allows you to perform some interesting algorithms, whether they relate to numbers, strings, or whatever complex structure you are working with.

Looking at our sample code:

```
{



{

<< << << __FUNCTION__ <<

myVector17 = { 4 };



auto addLeftVector =
```

```
auto multVector =

<< "The adding fold-left total is: " << addLeftVector << ", " <<

  << "and the multiplication fold total is: " << multVector <<

<<



//int x = ((((((0 * 10) + 1) * 10) + 2) + 3) * 10) + 4;

//int y = (((((((0 * 10) + 4) * 10) + 3) * 10) + 2) * 10) + 1;

auto MultiplyLeftFn = existing, v) { return (existing * + v; };

auto MultiplyRightFn = v, existing) { return (existing * + v; };

auto shiftLeftVector = MultiplyLeftFn);

auto shiftRightVector = MultiplyRightFn);

<< "The shifting fold-left total is: " << shiftLeftVector << " vs fold-right is: "
<< shiftRightVector <<

}

};
```

We first perform a which means you start with an initial value that you supply, and call the algorithm with each member of the range. Our example is simple, where we have an initial value of 0 and add each value of the range. You can see our result is 10 = 0 + 1 + 2 + 3 + We also use the std::multiplies with our initial value of 1. 1 * 1 * 2 * 3 * 4 = which we remember from our last topic is 4!

We then provide a couple of custom functions, and perform a ranges::fold_left() and just for comparison. We made it clear in our custom functions by showing that MultiplyLeftFn() multiplies the existing value and then adds our current iterator's value. MultipleRightFn() does the same action, but the parameter order is reversed, as shown:

```
Full range : 1, 2, 3, 4,
The adding fold-left total is: 10,
and the multiplication fold total is: 24

The shifting fold-left total is: 1234 vs fold-right is: 4321
```

Figure 13.9: Output from LookAtFolds

You can see the first result is since it executes as: ((((((0 * 10) + 1) * 10) + 2) + 3) * 10) + conversely, when it executes as (((((((0 * 10) + 4) * 10) + 3) * 10) + 2) * 10) + the total is

Aside from the function that you can create for this range-folding mechanism, you have multiple range functions that you can use, as shown in Table

Table 13.1: Comparing fold functions

With the ranges::fold_left_first() and ranges::fold_right_last() functions, remember that the return type is This might be because if your range is empty, then there is not even an initial value to return.

There are a couple of other variations of these range functions named ranges::fold_left_with_iter() and

Conclusion

It should be clear by now why we have so many chapters about ranges. They are versatile, and powerful, and there is still a lot more range functionality to talk about! Aside from retrieving simple min/max type functionality, we can also be given the iterators, which can provide extra insight. In terms of extra abilities, the sorting features for ranges are very good, aside from the basic sort, which is not basic if you add in your own comparer algorithm and custom projection functions. Then, we add in partial sorting, or merging, or a sort + copy combination, providing lots of flexibility as to how you can apply sorting capabilities to your data. The power of the fold and permutation functions makes things a lot cleaner in your code, it is much easier to call an existing library function than to create your own code.

Points to remember

Many of the range functions allow projections, which should save you steps in terms of custom loops.

The ranges library offers many functions that will not work properly if your container is not already sorted.

Executing a binary search on your range, regardless of base container type, is very powerful. Try to use that whenever you can, to speed up your code.

The fold functions are very handy, in that it performs an operation while modifying your aggregate value. You just have to make sure you select the right one.

References

(n.d.). Sort. Retrieved from https://en.cppreference.com/w/cpp/algorithm/ranges/sort

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 14

Range Algorithms: Memory and Modification Functions

Introduction

In our earlier chapters of this book, when we were talking about all the concepts, the topics seemed very different, and things did not start coming together until we started talking about topics in other chapters. Although the volume of chapters seems similar, these topics do not seem as abstract. With most of the functions we talk about, you are probably thinking I can use that right now! While this is true, and thankfully very useful, we are going to talk about some more complex data usage in our first couple of topics. We will then finish with some sequence functions that will be a good lead-in to our next chapter.

Structure

In this chapter, we will cover the following topics:

Recipe 14.1: Heap functions

Recipe 14.2: Uninitialized memory functions

Recipe 14.3: Modifying sequence functions

To help know if the feature works with the compiler you are using, we have included a list of the functions we will talk about, as well as an indicator whether it built and ran with each of the three main compilers.

```
{

// work with  compiler?   when tested  clang

LookAtHeap();                // 2023.10.22  yes

LookAtModifying_Copy();          // 2023.10.22  yes

LookAtModifying_Move();          // 2023.10.22  yes
```

```
LookAtModifying_FillAndRemove(); // 2023.10.22  yes

LookAtShiftAndRotate();          // 2023.10.22  no    no yes

LookAtReplaceAndSwap();          // 2023.10.22  yes

2023.10.22 yes  yes  yes


LookAtTransformAndGenerate();    // 2023.10.22  yes

}
```

Objectives

By the end of this chapter, the reader will understand a lot more about the operations that you can perform on a range. You will have learned about some of the basic copy and move functions, as well as the handy fill and remove functions. You will have also learned about the lesser-used shift, rotate, and shuffle functions, although they will be quite useful when needed. You will also learn about the transform and generate functions, which could be powerful tools when coding.

Recipe 14.1: Heap functions

Problem: We want some heap-related functionality that is easy to use.

Solution: The C++20 release has several heap functions that look at, and modify your range.

It might be helpful to some, to first talk about the heap data structure These are fairly simple, where any node will have at most, N number of children. There are different types of tree structures, and various sub-types as well, but overall, the heap is an important one. There are two main types of heaps, which are aptly named Min-heap and Max-heap. The name implies the focus of the tree, and how is it set up. For a Min-heap, the parent node has a smaller (or equal) value than the child. This would mean the root node has the smallest value. If you want the obtain the smallest, or smaller value in a Min-heap, it is fairly quick to do so. Conversely, if you have a Min-heap, and you want to obtain a larger value, then, relatively speaking, it will take a longer time. The other type is a Max-heap, which acts oppositely. The parent node has a larger (or equal) value than the child nodes, which means the largest value in a Max-heap is at the root node. Refer to the following table:

table:
table: table: table:
table: table: table: table:

Table 14.1: Small chart describing heap types

Since we normally engage with binary trees, we sometimes forget that they are an example of a heap structure. A binary tree is an example of a heap structure where you can have at most two children. They are still a type of heap, so would have the same characteristics of a heap, as we just talked about. Programming these structures and traversing them is sometimes one of the first things you do when learning a programming language.

When the values are stored in an array, the top value goes first, and then each tier is added, going from left to right. This consistency allows for assumptions and less work when adding or removing items from the heap structure. Refer to Figure

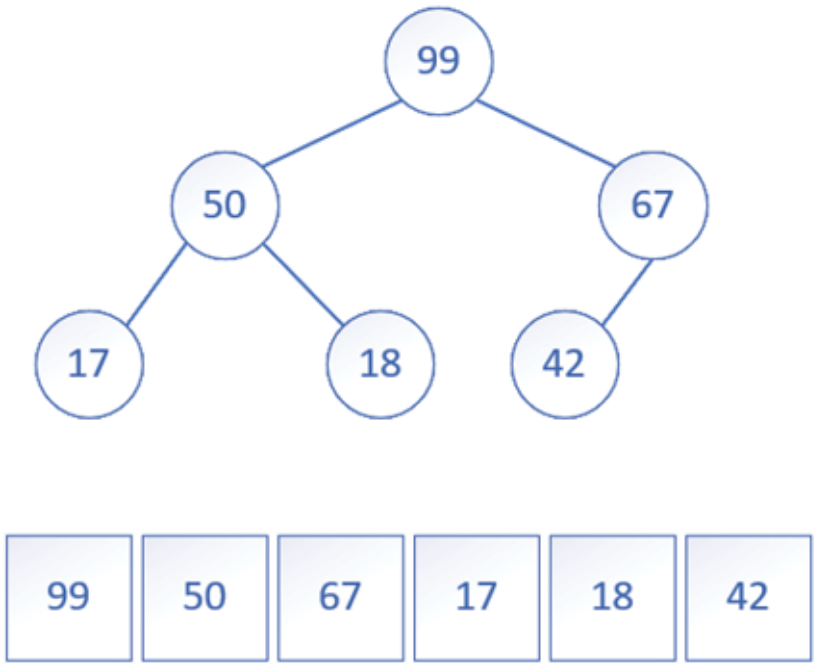

Figure 14.1: Binary heap and how values are stored in an array

This mechanism of storing the values in an array such that:

The largest value, the top of the binary heap, will be stored in the array's first position,

The next two largest values, the second row of the binary heap, will be stored in the next indices of the array, a[1] and

The next 4 largest (or 3 if the tree is unbalanced, like ours is in the preceding figure), will be stored in the next indices of the array, and a[5] (plus a[6] if it was a balanced tree). Note that the set of values in that tier { a[3], a[4], a[5], a[6] may not be sorted. But on the whole, that group of values are the next largest.

As the tiers get wider, possibly doubling in size, they will be the next group of values in the array.

We can show you another example looking at some code. We are still using the same headers as we did in Chapter and reusing our PrintRange() function.

```
{

myQueue = { 42 };

//std::ranges::sort(myQueue);

PrintRange(myQueue, "Original

=

<< "range" << (isHeapResult ? " is " : " is not << "a max heap" <<
```

```
<<


PrintRange(myQueue, "Max-heap

isHeapResult =


<< "range" << (isHeapResult ? " is currently " : " is still not << "a max heap"
<<

<<


PrintRange(myQueue, "Min-heap

isHeapResult =

<< "range" << (isHeapResult ? " is currently " : " is still not << "a min heap"
<<

<<

// go back to max-heap
```

```
PrintRange(myQueue, "Max-heap range, with extra value

<< "range" << ?

" is currently " : " is not << "a max heap" <<


PrintRange(myQueue, "Max-heap range, after

<< "range" << ?

" is currently " : " is still not << "a max heap" <<

<<

for = i >= --i) {


myQueue.begin()+i);

PrintRange(myQueue, "After pop# " + + ":

}

<<


PrintRange(myQueue, "Make into Max-heap range, one last
```

```
PrintRange(myQueue, "After
```

```
}
```

You can also follow the output result shown in Figure

```
Original range 18, 50, 99, 17, 67, 42,
range is not a max heap

Max-heap range 99, 67, 42, 17, 50, 18,
range is currently a max heap

Min-heap range 17, 50, 18, 67, 99, 42,
range is currently a min heap

Max-heap range, with extra value appended 99, 67, 42, 17, 50, 18, 77,
range is not a max heap
Max-heap range, after push_heap() 99, 67, 77, 17, 50, 18, 42,
range is currently a max heap

After pop# 7:  77, 67, 42, 17, 50, 18, 99,
After pop# 6:  67, 50, 42, 17, 18, 77, 99,
After pop# 5:  50, 18, 42, 17, 67, 77, 99,
After pop# 4:  42, 18, 17, 50, 67, 77, 99,
After pop# 3:  18, 17, 42, 50, 67, 77, 99,
After pop# 2:  17, 18, 42, 50, 67, 77, 99,
After pop# 1:  17, 18, 42, 50, 67, 77, 99,
After pop# 0:  17, 18, 42, 50, 67, 77, 99,

Make into Max-heap range, one last time 99, 67, 77, 50, 18, 17, 42,
After sort_heap 17, 18, 42, 50, 67, 77, 99,
```

Figure 14.2: Output from LookAtHeap

When we check our range originally, it says that it is not a max heap. That is resolved by one call to We verified this by calling There is also a function called ranges::is_heap_until() that looks at a subrange.

We also created a min heap, by passing std::greater() as the optional comparison argument. We then (after changing it back to a max heap) add a value to the end of our container, which of course now results in a non max heap. Calling ranges::push_heap() triggers it to look at that last item in the container, and figure out where it can be moved, in order to turn it back into a max heap.

The next bit of code iterates through a loop, once for each item in the container, but counting downwards. We call which looks at the begin and end iterators and moves the largest value to the end of the range. Since our loop is decreasing, we are looping to find the next highest value in the subset, and moving that to the end. The result of which is a sorted range, as the output shows. While that takes a few lines of code, it is interesting to see how it works. Of course, we can make it simpler by calling which takes the range from a max heap to a sorted range. You can see we get the same result from calling ranges::pop_heap() several times, as we did by calling

Recipe 14.2: Uninitialized memory functions

Problem: When dealing with ranges, we want a nice way to work with uninitialized memory.

Solution: C++ 20 has several functions relating to memory management.

Even with all of this modern C++ functionality, there are times when you need to work with allocating and deallocating raw memory. We will not go into the details here, because it does seem very specialized, and often the coding examples will get cumbersome with lots of casting involved. These functions are normally for those fringe cases, where instead of creating new objects directly with the new operator, you are allocating memory for the size of an object, and working with the memory that way. It is often done when we must align the data structure around a particular size of memory. There are already functions for this in the standard library (such as many of which were released with C++17), but in C++20, they created similar versions in the ranges library.

Some of the functions you could use are:

Copies a range of objects to an uninitialized memory. There is also which has a similar goal, but you provide a starting iterator and a count. If we think of the difference between and the copy() function knows the target memory is for that type of object, whereas unitialized_copy() has memory

allocated (such as with but the memory does not know what type of object that is supposed to go there.

Copies an object to uninitialized memory. There is also which has a similar goal, but you provide a starting iterator and a count.

Moves a range of objects to uninitialized memory. There is also which has a similar goal, but you provide a starting iterator and a count.

Constructs objects in uninitialized memory. Following the normal pattern, there is also

Instead of using the default constructors that were used with this uses value initialization. (If you had value types like int in your structure, then the default-construction does not set those variables. You can think of value-initialization as zero-initialization, where variables such as integers would be set to 0.) There a also

Creates an object at a specified address.

Destroys a range of objects. Aside from there is also which destroys an object at a specified address.

Recipe 14.3: Modifying sequence functions

Problem: We want lots of functionality for modifying ranges.

Solution: C++ 20 provided plenty of functions for you.

When it comes to manipulating a range, we touched upon it in the last chapter with a couple of the sorting functions, and earlier in this chapter with the heap functions. However, we have not been modifying a range that much. That is the focus of this topic, dealing with a number of ways we can move, or modify the items in the range.

Our first code example will look at copying into the range:

```
{

myNumberQueue = { 99 };

myNumberVector;

PrintRange(myNumberQueue,

PrintRange(myNumberVector, "Target before copy:



PrintRange(myNumberVector, "Target after copy :
```

```
<<

myNumberVector.clear();

= n) { return n % 3 == };

IsMultipleOfThree);


PrintRange(myNumberVector, "Target after copy of multiples of 3 :

<<

myWordQueue = { "functionality" };

"agree" };

PrintRange(myWordQueue, "Original

- 1 + introWords->length()

myStringVector.end());

myStringVector.begin());

PrintRange(myStringVector, "Copied (backwards)

}
```

Our first code block performs a simple although do not forget you can also use a predicate and a projection, to tailor it to your needs. We then use ranges::copy_if() to copy into our destination range if they are a multiple of 3. The output in Figure 14.3 shows that 4 of our original range met that criterion. With the last code block, we used ranges::copy_backward() to update our target range. The tricky part about this way was that we had to pre-size our destination. Note that it does not reverse the input range, but iterators from source[n] to source[0] copying the value to destination[n]. In our case, we sized the destination to hold a few extra words, and then performed a ranges::copy_n() to copy another range into the beginning of our destination.

```
Source 99, 17, 37, 75, 34, 67, 42, 99,
Target before copy:
Target after copy :  99, 17, 37, 75, 34, 67, 42, 99,

Target after copy of multiples of 3 :  99, 75, 42, 99,

Original sentence the C++23 release has a lot of range functionality
Copied (backwards) sentence Don't you agree the C++23 release has a lot of range functionality
```

Figure 14.3: Output from LookAtModifying_Copy

It is worth mentioning again how convenient it is that you can copy values from one range to another, without being concerned about what the underlying containing types they are.

The next code example we will look at deals with moving the ranges. The example uses strings, and so to show that we have empty strings (instead of an empty area), we change our PrintRange() function to allow a separator between our string values, and whether to quote the strings.

```
title, auto r, sep, bool printQuotes) {
```

```
if {

<< title << "

}

else {

<< "Full range :

}

if == {

<< sep;

}

=

for i< rCount; i++) {


if <<

<<

if <<
```

```
if << sep;

}

<<

}

auto r, title = {

using R =

using Elem =

", " };

if {

sep = ",

}

else {

sep = "

}
```

```
PrintRange(title, r, sep,

}

{

yesterdaysMenuChoices = { "roasted "spicy chicken" };

todaysSoupChoices;

dishes (before) yesterdaysMenuChoices,

soup choices (before) todaysSoupChoices, ",

<<



dishes (after) yesterdaysMenuChoices, ",

soup choices (after) todaysSoupChoices,

<<


specialMenu{ "Whatever else people didn't eat" };

todaysSoupChoices.end(), todaysSpecialSoupChoices.end());
```

```
specialMenu.end(), todaysSpecialSoupChoices.begin());

special soup choices todaysSpecialSoupChoices,

}
```

Similar to the ranges::copy() command, the ranges::move() function makes it easy to move values between ranges. There is a ranges::move_backward() function as well, that iterates through the source range and moves the value starting at the end of the target. You can see in the following output that we did a similar task of inserting another value to the beginning of our target. This time we used the begin() and end() iterators of the source, and the begin() iterator of the target, to say that is where to start. Refer to Figure

```
Yesterday's dishes (before) :
'roasted potato'
'broccoli'
'spicy chicken'
Today's soup choices (before) :

Yesterday's dishes (after) : '', '', ''
Today's soup choices (after) :
'roasted potato'
'broccoli'
'spicy chicken'

Today's special soup choices :
'Whatever else people didn't eat'
'roasted potato'
'broccoli'
'spicy chicken'
```

Figure 14.4: Output for LookAtModifying_Move

The ranges::move_backward() could be very handy if you are moving values within the same container. In Figure you will see that if we want to move (or copy) some text over to the right, then we will overwrite our value. You could use a temporary buffer, but it is easier to use ranges::move_backward() to solve the problem.

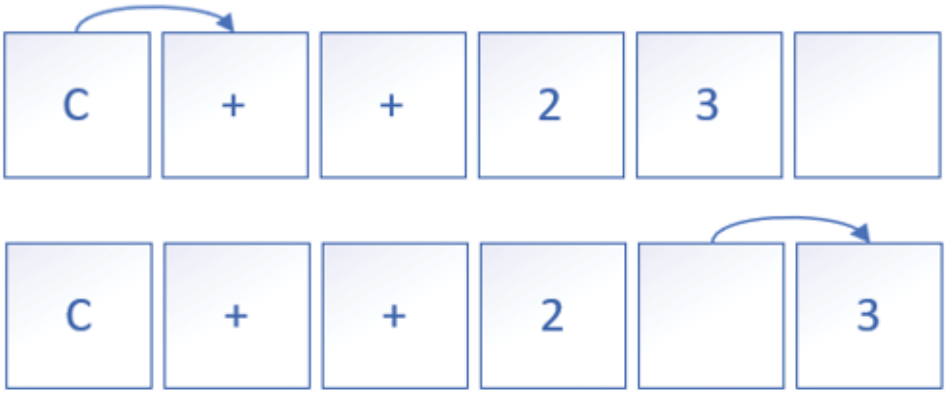

Figure 14.5: Diagram showing ranges::move_backward()

The next code sample shows which can fill range with a value. There is also which takes an iterator and a length. The following code block uses ranges::fill() to fill the range with exclamation marks. Then it uses ranges::fill_n() to fill the rest of the character range with characters from a string. We then use ranges::remove() to remove the exclamation mark, since it is looking for a value to remove not an offset. We then use a predicate to remove all the vowels and spaces from our character range. After calling the message may not be readable anymore, but at least it might fit onto an automobile license plate:

```
{



source{ "C++23 is great" };



message, "

auto it = message.begin();
```

```
for = i < source.size(); i++, it++) {

source[i]);

message, "



}

<<

=

after removeResult.begin()), "

message.erase(removeResult.begin(), removeResult.end());

auto IsVowelOrSpace = c) { return c == ' ' || c == 'a' || c == 'e'

  || c == 'i' || c == 'o' || c == };

removeResult = IsVowelOrSpace);

after removeResult.begin()), "

<<

message.clear();
```

```
target;

IsVowelOrSpace);

after message, "

after target, "

}
```

Figure 14.6 shows the result:

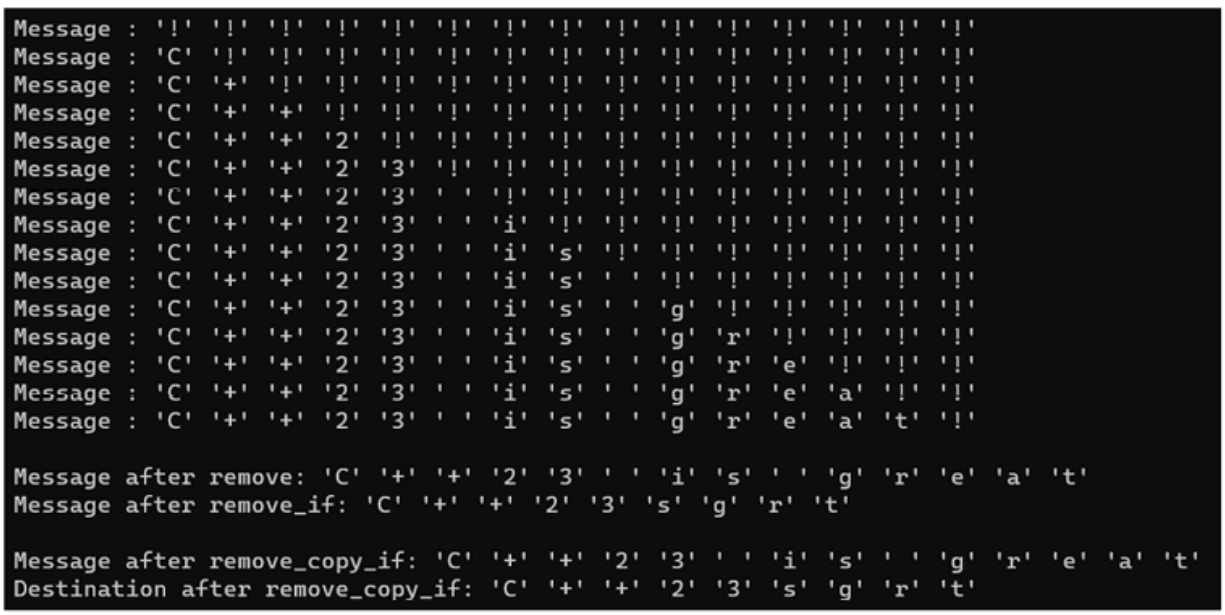

Figure 14.6: Output from LookAtModifying_FillAndRemove

If you look at the last code block, we reset our message range and then call Instead of removing values from the source range, this copies the values to a target range. You can see the characters match our previous call to except the values are copied instead of removed. It does seem a bit odd to have since it does not actually remove values from the range. Moreover, it is very similar to the difference being that copy_if() will copy the value if the predicate returns true, whereas remove_copy_if() will copy the value if the predicate returns false. They probably created ranges::remove_copy_if() since they already had the similar function in the standard library, and wanted to keep things consistent.

Instead of moving values from one place to another, we will now look at shifting a value, either using ranges::shift_left() or The main thing to be careful of is depending on the data type your container is holding, it will determine what happens when values get shifted. If you look at the code, and the output, you will see that for containers that hold simple types like char or int, the values do shift, but the old value will exist in the old spot as well. If it is a struct, or in our case, a string, then when it shifts, the old item will now point to nothing.

{

<< << << __FUNCTION__ <<

#ifdef __cpp_lib_shift

message;

is

```
message, "

after shift message, "

after shift message, "

after shift left message, "

after shift right message, "

<<

myNumberQueue = { 99 };

PrintRange(myNumberQueue, "Original

PrintRange(myNumberQueue, "After shift_left

PrintRange(myNumberQueue, "After shift_right

<<
```

```
words = { "!" };

PrintRange(words, "Original


shift_left words, "


shift_right words, "

<<

textMessage{ "great!" };

text textMessage, "


message rotated textMessage, "

textMessage.end() -


message rotated textMessage, "

textMessage.begin() +

message rotated textMessage, "
```

```
<<

target;

textMessage.begin() +

target, "

#endif

}
```

You will note that the function has an #ifdef As mentioned earlier, a number of recent features have added a definition, so you will know whether it has been implemented. The one problem in this case, at the time of writing, the gcc and clang compilers have implemented the definition, but not the actual feature.

The key part is, when values are shifted out of the range, then that value is gone. If, instead of shifting it into oblivion, you prefer the value wraps around to the other side, then you can use Our code block shows us rotating +1, -3, and then +2, the net result is our strings are in the same order as we began. The parameter for the rotate functions accepts an iterator. You are specifying the value of that iterator should be the new first value of the range. As with other functions we have seem, ranges::rotate_copy() does not alter the source range, but copies the values to a new range. In our case, we told it to start on the second element and the copying wraps around until it gets back to where it started.

Refer to Figure

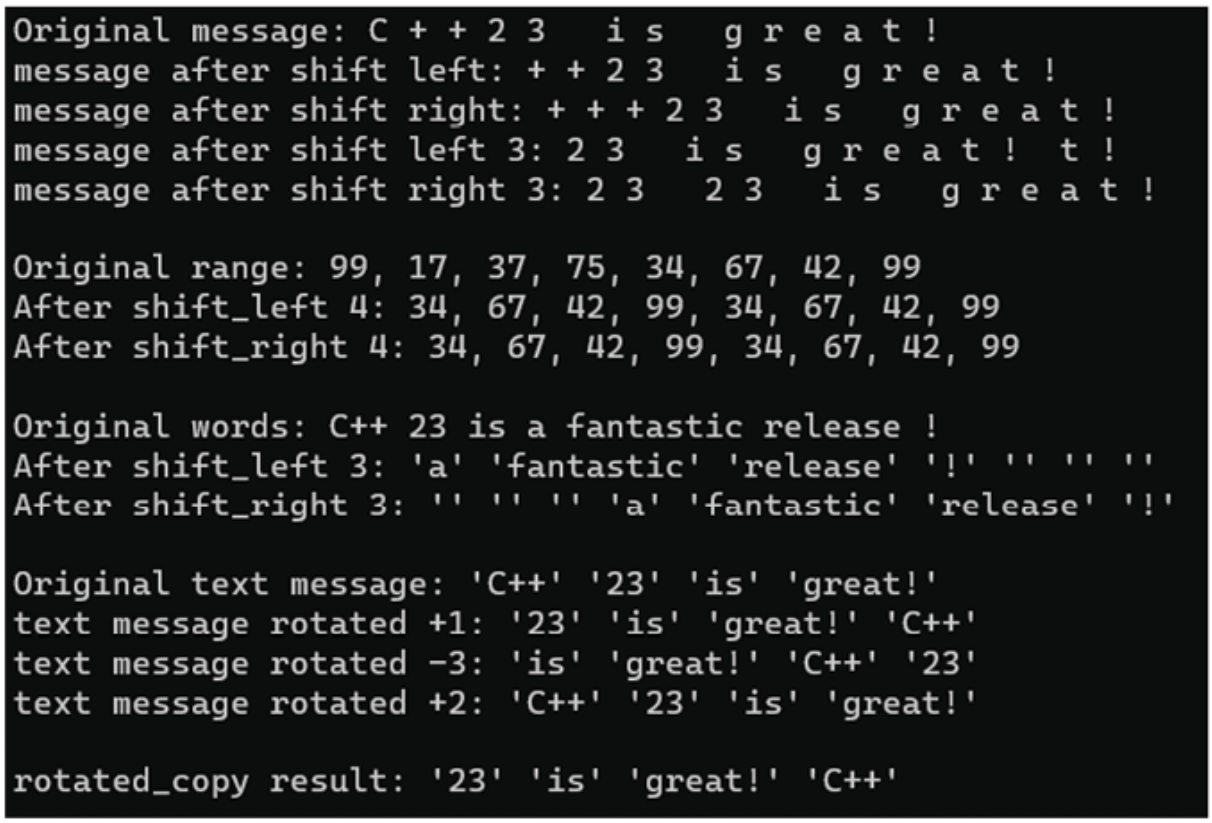
Original message: C + + 2 3 i s g r e a t !
message after shift left: + + 2 3 i s g r e a t !
message after shift right: + + + 2 3 i s g r e a t !
message after shift left 3: 2 3 i s g r e a t ! t !
message after shift right 3: 2 3 2 3 i s g r e a t !
Original range: 99, 17, 37, 75, 34, 67, 42, 99
After shift_left 4: 34, 67, 42, 99, 34, 67, 42, 99
After shift_right 4: 34, 67, 42, 99, 34, 67, 42, 99
Original words: C++ 23 is a fantastic release !
After shift_left 3: 'a' 'fantastic' 'release' '!' '' '' ''
After shift_right 3: '' '' '' 'a' 'fantastic' 'release' '!'
Original text message: 'C++' '23' 'is' 'great!'
text message rotated +1: '23' 'is' 'great!' 'C++'
text message rotated -3: 'is' 'great!' 'C++' '23'
text message rotated +2: 'C++' '23' 'is' 'great!'
rotated_copy result: '23' 'is' 'great!' 'C++'

Figure 14.7: Output

The next family of functions to look at, relate to replacing values in your range. These functions follow the same pattern as we have seen so far. Looking at the following code, you will see that ranges::replace() replaces a value with another for a given range. There are, of course, overloads that you can provide a projection, as well as an iterator and sentinel if you do not want to perform the replace on all items of your range. There is also which takes a predicate and if that call returns true, will perform the replacement. We also have which does not modify the source range but copy the values into a provided target. As we have seen before, the downside to this is we need to pre-size the destination.

```
{


myNumberQueue = { 99 };

PrintRange(myNumberQueue, "Original


PrintRange(myNumberQueue, "Numbers after

n) { return n % 2 == };

IsEven,

PrintRange(myNumberQueue, "Numbers after

<<

myNumberQueue = { 99 };


target.begin(),

PrintRange(target, "Destination after

target.clear();

target.resize(myNumberQueue.size());
```

```
target.begin(), IsEven,

PrintRange(target, "Destination after

<<

message{ "& "release!" };


menuIdea{ "to make "sauce" };

message, "

menu menuIdea, "

+ message.begin() +

+ menuIdea.begin() +

after message, "

idea after menuIdea, "

<<

}
```

The last code block has interesting uses; with ranges::swap_ranges() you provide the ranges (or begin and end iterators) and it performs a swap between the two ranges. Obviously, they have to be the same size and compatible types. You can see we swapped words between our message and some menu instructions.

```
Original numbers: 99, 17, 37, 75, 34, 67, 42, 99
Numbers after replace: 66, 17, 37, 75, 34, 67, 42, 66
Numbers after replace_if: 1, 17, 37, 75, 1, 67, 1, 1

Destination after replace_copy: B ◄ % K " C ★ B
Destination after replace_copy_if: c ◄ % K ☻ C ☻ c

original message: 'C++' '23' 'is' 'a' 'great' '& exciting' 'release!'
original menu idea: 'to make your' 'linquine' 'very' 'tasty' 'add' 'a' 'creamy' 'sauce'
message after swap: 'C++' '23' 'is' 'a' 'very' 'tasty' 'release!'
menu idea after swap: 'to make your' 'linquine' 'great' '& exciting' 'add' 'a' 'creamy' 'sauce'
```

Figure 14.8: Output from LookAtReplaceAndSwap

There is also a ranges::swap() function, but we will cover that in an upcoming chapter, since it is deemed as more of a utility function.

Our next code example involves some handy functions. The code example lists the order of which utensils you would use at a very fancy dinner. If you wanted to ignore etiquette and make people gasp, then you could use your utensils in the reverse order that you should. After we call you can see in the output that you can start with your dessert fork! As usual with these functions, we can also perform ranges::reverse_copy() that updates another range and does not alter the original.

{

utensils{ "butter "oyster "salad

```
"soup and "dessert fork"

utensils, ",



utensils, ",

target;



after target, ",

<<

courses{ "hors "main "palate



of 8 course courses, ",



ranGen{ rd() };

for = i < i++) {

ranGen);

after shuffle#" + + courses, ",
```

```
}

<<

courses = { "hors "main "palate };

target.clear();

for = i <= i++) {

target.clear();

i, ranGen);

after sample#" + + target, ",

}

<<

manyUtensils{ "spoon" };

utensils  : manyUtensils, ",

moreUtensils;

// to use later
```

```
[dupFirst, dupLast] =

manyUtensils.erase(dupFirst, dupLast);

manyUtensils, '',

target.clear();



after target, '',

}
```

We then take our list of eight dishes for our fancy dinner, and randomize then with We print the new sequence out a handful of times, and you can see in the following Figure 14.9 which one sounds best. When we use it takes existing range items and rearranges them. We also call but that takes random range items and copies it do another range. (It almost could have been called In the last code block, we have a bunch of utensils, but after calling we got rid of the duplicates and are just down to a few. Note that you must have the items sorted before calling which we did by calling ranges::sort() first. We also called where the unique result is copied into another range.

```
original order: butter knife, oyster fork, salad fork, soup spoon, knife and fork, dessert fork
after reverse: dessert fork, knife and fork, soup spoon, salad fork, oyster fork, butter knife
target, after re-reverse: butter knife, oyster fork, salad fork, soup spoon, knife and fork, dessert fork

order of 8 course meal: hors d'oeuvre, soup, appetizer, salad, main course, palate cleanser, dessert, mignardise
course after shuffle#1: soup, hors d'oeuvre, mignardise, palate cleanser, dessert, main course, salad, appetizer
course after shuffle#2: main course, appetizer, salad, palate cleanser, hors d'oeuvre, dessert, soup, mignardise
course after shuffle#3: dessert, mignardise, palate cleanser, hors d'oeuvre, salad, soup, main course, appetizer
course after shuffle#4: soup, main course, hors d'oeuvre, dessert, salad, appetizer, palate cleanser, mignardise
course after shuffle#5: salad, appetizer, palate cleanser, soup, hors d'oeuvre, main course, dessert, mignardise

course after sample#1: hors d'oeuvre
course after sample#2: soup, appetizer
course after sample#3: hors d'oeuvre, soup, main course
course after sample#4: hors d'oeuvre, soup, salad, mignardise
course after sample#5: salad, main course, palate cleanser, dessert, mignardise

duplicated utensils   :  knife, fork, fork, spoon, knife, spoon, knife, fork, spoon
de-duplicated utensils: fork, knife, spoon
utensils after unique_copy: fork, knife, spoon
```

Figure 14.9: Output from LookAtReverseShuffleUnique

Let us now move on to the last code example. We have manipulated the range in a number of ways: we moved, reversed, sorted, or randomized it. That was mainly positional, but with many of the range functions, we have seen where we can use a predicate to actually change the value. We also have the ranges::transform() function which has that as its primary task.

{

{ value *= return value; }

1 };

};

{

```
<< << << __FUNCTION__ <<


term1, '',

term1.end(), term1.begin(),

c) -> { });


after term1, '',

<<

auto TextPadder = c) -> { " + + " };

term2;

TextPadder);

term2, '',

<<

keys{ "i" };

values{ "r" };

dictionary;
```

```
= k, v)

  ->

};

values, dictionary.end()), MakePair);

<< "Created map:" <<

for : dictionary) {

<< "Key=" << kvp.first << ", " << kvp.second <<

}

<<



for = i < numbers.size(); i++) {

NumberDoublingGenerator());

}

PrintRange(numbers, "Generated
```

}

Our code shows a few ways you can use The first is a simple transformation where we change it to uppercase, and the output goes back into the same range. The next code block is using that same range of characters, and this time when we transform it, we change it to a different type and store that in a different range. Then we have an interesting code block that takes two different ranges (one of keys, and the other of values), and our transform takes those values and calls std::make_pair() and that result gets outputted into a map, as shown in Figure

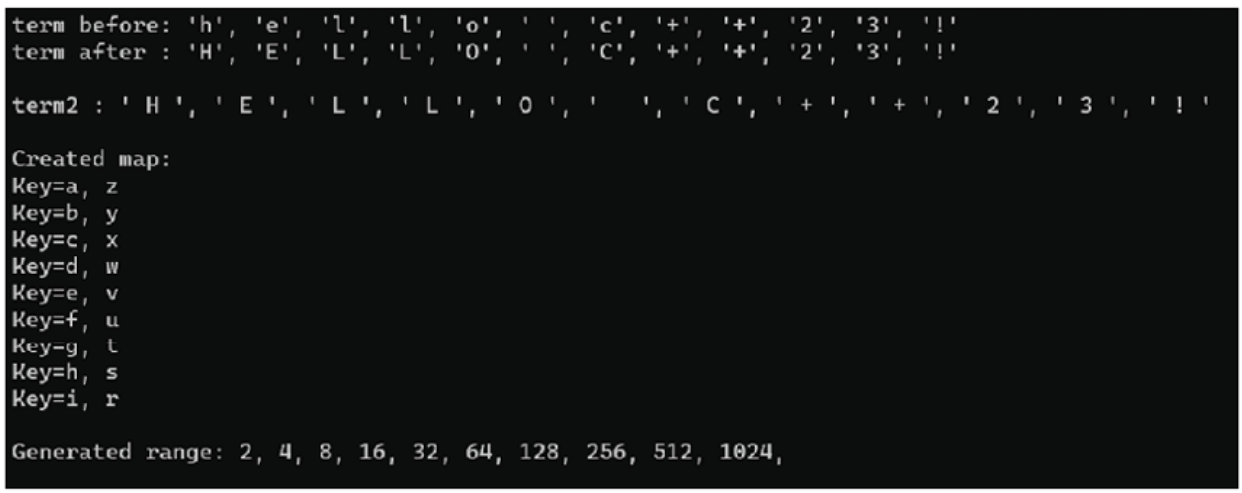
term before: 'h', 'e', 'l', 'l', 'o', ' ', 'c', '+', '+', '2', '3', '!'
term after : 'H', 'E', 'L', 'L', 'O', ' ', 'C', '+', '+', '2', '3', '!'
term2 : ' H ', ' E ', ' L ', ' L ', ' O ', ' ', ' C ', ' + ', ' + ', ' 2 ', ' 3 ', ' ! '
Created map:
Key=a, z
Key=b, y
Key=c, x
Key=d, w
Key=e, v
Key=f, u
Key=g, t
Key=h, s
Key=i, r
Generated range: 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024,

Figure 14.10: Output from LookAtTransformAndGenerate

The last function to look at in this chapter is The focus is not on the range as much, but the callable you provide to it (whether it be a lambda, or in our case, a class with an operator() overload). This allows you to fill a range by calling a function that will generate a value. In our case, we are doubling a number each time we call our function, and it gets stored in the range. There is also a ranges::generate_n() if you want to be more explicit on how many

times it gets called. We will be looking at more generate/generator functionality in the next couple of chapters.

Conclusion

We did say that ranges were a very big part of C++20, both in terms of importance, and but also in terms of its volume of functionality. We have covered a lot of ground, but we still have some big ranges features to talk about. We have seen that ranges can be used to think about heaps and maximize the performance of finding a value in a range. We have looked at a lot of coding examples dealing with manipulating a range; whether that means moving values around, or transforming values. All of this functionality will come in handy when we learn more about other range features.

Points to remember

Most range features were in the C++20 release, and many of those have similar counterparts in the standard library.

Heaps could be a handy tool if you want to investigate binary trees and/or how to find data quickly.

Most of the range modifying functions come with different versions; many have an _if version, so your predicate can determine if you should take the action. Many also have _copy versions which will leave your source range unmodified and instead, it outputs to another range. Just be careful that some of these functions require the target range to be pre-sized.

The ranges::copy_backward() and ranges::move_backward() are useful when transferring the value within the same range. You may not have to use temporary variables since you will not walk over the same item.

Think of your data type when using ranges::shift_left() or Your original value could get nullified, or it could still be there.

Remember the function ranges::shuffle() rearranges the items within a range, whereas the ranges::sample() function copies shuffled items into a new range.

The result of ranges::unique() is a subrange to where the duplicated values got moved to. You will need to erase that to actually get the duplicates out of your container.

The ranges::transform() function is powerful, allowing you to execute a niebloid on one range and store the result in another range. It might become one of your favourite ranges tools.

References

Wikipedia. (n.d.). Heap. Retrieved from https://en.wikipedia.org/wiki/Binary_heap

Join our book’s Discord space

Join the book’s Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 15

Views and Range Adaptors

Introduction

Similar to how C++20 introduced how you can create your own concepts, but still gave you lots of built-in ones to use, ranges::views are new. However, C++ has shipped with lots of newly created views that will help your code out tremendously. A few chapters previously, we learned what views are, but now we will continue that journey and see what views exist in the ranges library. Moreover, we will talk about what one could say is a view's best feature, that is, its composability.

Structure

In this chapter, we will cover the following topics:

Recipe 15.1: Review of views and ranges, and what are range adaptors

Recipe 15.2: Some simple range adaptors

Recipe 15.3: Using take() and drop() adaptors

Recipe 15.4: Using range adaptors with composition

Recipe 15.5: Using filter() and transform() adaptors

Recipe 15.6: Working with the join() and split() adaptors

Recipe 15.7: Working with the keys() and values() adaptors

Recipe 15.8: Using chunk() and slide() adaptors

Recipe 15.9: Using zip() adaptor

Recipe 15.10: A quick look at reverse()

Recipe 15.11: Looking at special adaptors

Some good news about these features; although the compiler makers were slow to implement these features when they were first released, things are working well. The small caveat below, was the gcc compiler was having issues when the zip code was being optimized.

{

}

Objectives

By the end of this chapter, the reader will understand about what views exist in the ranges library, and how to join them together.

Recipe 15.1: Progressing from views and ranges, to range adaptors

Problem: We should know what a range adaptor is before we try to use it.

Solution: A range adaptor was made to convert a range into a view.

We have been talking a lot about ranges lately, but it has been a while since we have talked about views. A range can be defined as an ordered sequence of elements, that has a designated start, and a rule for A view is syntactically defined as a concept that is:

a range

is std::movable (although in the early stages, it was sometimes defined as semiregular, which mainly meant that it was copyable)

has a Boolean property named enable_view (which is mainly just a flag to designate it as a view)

A view is also semantically defined as:

constant time copy operation

constant time move operation

constant time assignment

constant time destruction

The enable_view flag can be used to opt out of being a view, if you meet all the syntactic requirements, but do not meet all the semantic requirements.

The next important part to recall about views is that they are handled lazily. While we have seen that all the functions that work with a range, work on the data right away, a view does not work on the data until it is told it needs a value.

When you are working with views, remember that we have been focusing the last few chapters on range algorithms. Those were created to work on ranges, whereas the focus for range adaptors is on views, which is what we will be looking at in this chapter. Range adaptors are composable, range algorithms are not.

We will be using a lot of views in the range's namespace. Those entities are types, and many have a corresponding object in the view’s namespace. For example, there is the ranges::take_view type, and the views::take object, which is normally called a range adopter object.

A range adaptor is something that takes a range and can convert it into a view. The range adaptor (normally) does not just create a view of the same data, but it transforms the range of data. That is not too hard to imagine, because we have seen with ranges that we can call a projection, which we can think of as a function, which changes the value, or even instantiates a new type. Dealing with ranges, we have used subranges a few times.

These are partial ranges, which also is not difficult to picture, since we just said a range has a defined start, and a sentinel to state when we are at the end. So, a view is a range, that possibly has had a projection done on it, and possibly has had the beginning and end redefined.

If we want to get a bit deeper into it, some ranges functions returned a subrange, which we just said was not too difficult to define. A view adaptor would return something that is more complex. For now, we can say it returns a closure, it is not just data, but a larger type that could have some temporary variables that are used when it is being lazily evaluated.

Recipe 15.2: Some simple range adaptors

Problem: To learn more about range adaptors, what are some of the simpler ones existing?

Solution: Using an all-or-nothing approach, we will start with and and then go from there.

In the previous topic, we said a range adaptor could use a projection to get a new value, and it could change the beginning and end point for the range. If it did neither of those things, we would get the same data, which is what views::all() has.

Our code sample has lots of header includes, and we do not need them all now of course, but it is easier having them all together than adding them as we progress in this chapter:

```
#include

#include

#include

#include

#include
```

```
#include

#include

#include

#include

#include

#include

#include

#include

#include

#include

#include

#include

#include
```

```
#include

#include

#include

#include

#include

#include

#include

#include
```

Further, we have our code to look at:

```
{


title, auto view) {

<< title <<

<< "Distance= " << view.end())
```

```
  << ", size=" << view.size() <<

<< "View is " << (view.empty() ? "empty" : "not <<

<<

}

{

const severalNumbers{ 42 };

=

allResult);



emptyResult;

emptyResult);

singleResult{ 42 };

singleResult);

}
```

```
};
```

You can see that we call views::all with our vector, and then we pass that to a function that outputs the size. We then call the same function with an instance of If you are wondering why they created a type that holds nothing, then the best reason is probably what we have shown in our example. If you are testing a view, and want to ensure you can handle a case where the view has no values, then empty_view<> seems like a good way to test things, as shown in Figure

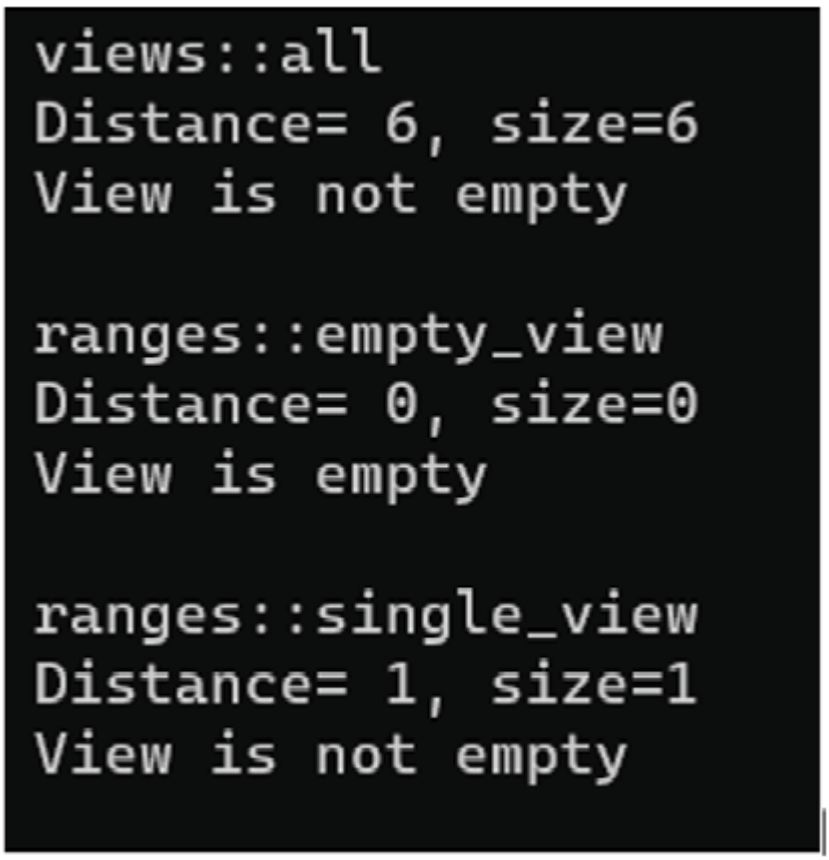

Figure 15.1: Output from LookAtAllAndEmptyAndSingle

If you wanted to ensure how a test worked if you had just one value, then you could use Technically, they categorize ranges::single_view<> and ranges::empty_view<> as factories (which we will be looking at in our next chapter), but it seemed like an easy way to get started. The other technical tidbit is that we said most views do not own the data they are looking at, but ranges::single_view<> is one of the exceptions. Since they figured it was only one value, it is easier for it to own that value.

One last thing to mention is that many of the view adaptors we will be looking at derive from but the three we just looked at did not. Luckily, we have concepts and we could declare our function using the ranges::view concept. The following are functions defined in but the other interesting thing is that it normally uses Curiously Recurring Template Pattern which we talked about in an early chapter. The member functions are:

Boolean stating if it has any values. There is also the operator bool function, which states if not empty.

returns the number of elements.

returns the first element of the view.

returns the last element of the view.

returns the element of the view.

returns the address of the view’s data.

Of course, with all of the functions, there are the caveats that for the function to work, they must satisfy the appropriate range concept.

Recipe 15.3: Using take() and drop() adaptors

Problem: We want a way to take only some of the items from the view.

Solution: C++ 20 has functionality for take() and

We have seen the all() view adaptor where it keeps the entire range that it was given. Now, we will look at some other adaptors that keep only some of the range's values. In the following code example, we will first look at This is simple enough that you provide a range (that is a container of values), and the number of values you want to keep. It will always start at the beginning of the values. Our next code block uses a variation of that, named Contrary to this does not take/keep a specific number of values, but it will continue to take values while the specified predicate returns true for the value it looks at. In our case, we provided an IsEven() predicate, and it takes the first four numbers since they are even.

You can see in the output shown in Figure that it prints out the values until it gets to the odd value 99. Our next code block looks at a related function, named Instead of keep the first N values, it drops/skips/ignores the first N values. Our code says to drop the first 7 values, so it prints everything from the onward. Moreover, in this family of functions is You could guess that if take_while() kept the values as long as the predicate accepts the value, drop_while() skips over the value as the predicate returns true for the value. In our output, while take_while() prints until it sees the value 99, drop_while() does not start printing until it sees the value 99. In our

next code block, we combine a couple of functions. We embed the drop() function into so the output of drop() is fed into The output shows we dropped the first 7 values, then take the next 3.

Refer to the following code example for a better understanding:

```
{



title, auto view) {

= n) { << n << ", };

<< title << ":

for : view) {

PrintNumbers(i);

  }

<<

}

};

{
```

```
{

<< << "Examples in: " << __FUNCTION__ <<

= n) { return n % 2 == };

const severalNumbers { 25 };

  severalNumbers);

const auto someAtStart =

  someAtStart);



const auto someEvenAtStart = IsEven);

  someEvenAtStart);

const auto someAfterStart =

  someAfterStart);

const IsEven);

  someOddAfterStart);
```

```
const auto dropSomeAndTakeSome

  7 and take dropSomeAndTakeSome);

const auto it99 =

const auto someAfter99 =

  after someAfter99);

const auto someAfter99_2 +

  after 99 someAfter99_2);

<<



}

};
```

That is the power of these adaptors: we can combine them easily and do not need extra variables. You will notice we also created a common utility class to print our view since we will be using that for several examples.

You will see in our last code block it shows another function, named This does something similar to our drop-and-take approach. Firstly, we call

ranges::find() for a value, and we pass that to counted() with an argument saying how many we will take. The call after that to counted() skips a step. If you know the index to use (we said we would use then we could just use that.

Refer to Figure

```
all: 4, 42, 2, 22, 99, 37, 17, 67, 14, 27, 33, 7, 6, 65, 25,
take 5: 4, 42, 2, 22, 99,
take_while even: 4, 42, 2, 22,
drop 7: 67, 14, 27, 33, 7, 6, 65, 25,
drop_while even: 99, 37, 17, 67, 14, 27, 33, 7, 6, 65, 25,
drop 7 and take 3: 67, 14, 27,
4 after 99: 99, 37, 17, 67,
4 after 99 (again): 99, 37, 17, 67,
```

Figure 15.2: Output for LookAtTakeAndDrop

Recipe 15.4: Using range adaptors with composition

Problem: We want a cleaner way of combining adaptors.

Solution: Instead of embedding the calls to the range adaptors within one another, C++ 20 also has an interesting way of combining them.

While it is good that we can combine range adaptors, you could imagine that if we wanted to combine several, it would make the code a lot less readable. Therefore, when ranges and views were first released, they also created a new symbol and syntax with the pipe '|' operator. For those that know Linux/Unix, you should be used to this symbol. It allows you to send the output of one command as input to another command. This is basically the same as what we are trying to accomplish here, so although the syntax is different than what we are used to, for some it seems very natural.

If we redo our previous code sample, this time using composition, you can decide for yourself which seems cleaner:

```
{



{

<< << "Examples in: " << __FUNCTION__ <<
```

```
= n) { return n % 2 == };

const severalNumbers { 25 };

  severalNumbers);

const = severalNumbers



  someAtStart);

const = severalNumbers

  someEvenAtStart);

const = severalNumbers

  someAfterStart);

const = severalNumbers

  someOddAfterStart);

const = severalNumbers |

  7 and take dropSomeAndTakeSome);
```

```
const auto someAfter99 = severalNumbers n) { return n != })

  |

  after someAfter99);

}

};
```

You can see when we first call we are piping the contents right into the view. At first, it looked odd, wondering how the container knew what to do next. But when they are chained like that, it is more so that take() sees that there is input that it can use. The next few examples follow the same approach; it is not a lot less typing, but it is interesting to see that these adaptors do not need that extra parameter and can just use the input that is being fed into them. Whereas before, when we combined a couple of functions, even that started to get a bit messy. Now when we pipe the container into drop() and pipe that into not only does the code seem cleaner, but the logic of what we are trying to do seems more clear as well. Although we could have, we did not compose the same call as we did with the previous recipe. Here, we dropped the input until we got to the value 99, and then we took the next 4. The following output shows the same result as before, but comparing the two code examples, we certainly like the latter one.

```
all: 4, 42, 2, 22, 99, 37, 17, 67, 14, 27, 33, 7, 6, 65, 25,
take 5: 4, 42, 2, 22, 99,
take_while even: 4, 42, 2, 22,
drop 7: 67, 14, 27, 33, 7, 6, 65, 25,
drop_while even: 99, 37, 17, 67, 14, 27, 33, 7, 6, 65, 25,
drop 7 and take 3: 67, 14, 27,
4 after 99: 99, 37, 17, 67,
```

Figure 15.3: Output from LookAtTakeAndDropWithComposition

Recipe 15.5: Using filter() and transform() adaptors

Problem: We want to have the power to transform values, as we did when working with ranges.

Solution: C++ 20 has views::transform() to change values, and views::filter() to use only particular values.

Hopefully, when we were working with ranges, you could appreciate how we could project (in other words alter) the values. The same power exists with ranges::transform() when we made new instances of another type. We can do the same thing with views, via the transform_view() range adaptor. We can also easily filter so that we are only looking at values we want, using This is similar to the ranges::remove_if() algorithm we covered in the previous chapter. The main difference is that with the algorithm, we were removing the value if we did not want it; now we are filtering to only keep the values we want.

In the following code example, we are filtering to keep the odd numbers, storing that in the justOdd variable and printing them out. The next code block shows us compose that we are first filtering to keep the odd numbers, then we call transform() to double the values. We then do something interesting; we pass the justOdd result we had before, that we already printed, and pass that into the transform() adaptor.

```
{
```

```
{

<< << "Examples in: " << __FUNCTION__ <<

= n) { return n % 2 == };

auto Doubler = n) { return n * };



const severalNumbers { 25 };

  severalNumbers);

= severalNumbers |

  justOdd);

const = severalNumbers |

  odds are doubled doubleOdds);

const = justOdd

  odds justOddsDoubled);

const message = "C++23 is

= c) -> {
```

```
if "" + c) +

"" + c) +

"" + c) +

  };

const = message

for {

<< s <<

  }


}

};
```

You can see that we got the same output, despite doing it differently, by reusing previous output.

In the next code block, we are taking a which, if you recall, can be interpreted as a range, or a view. We take our range of characters and call a transformation. This takes each character and returns a string, showing that we can still transform from one type to another.

Refer to the following figure:

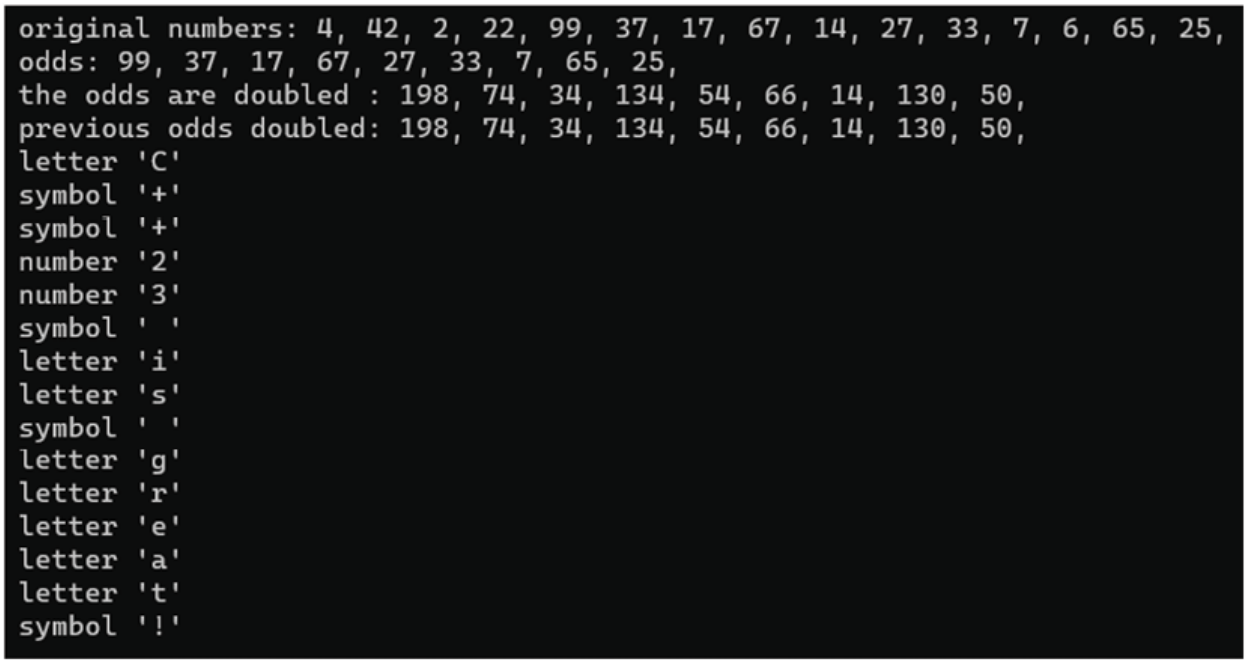
original numbers: 4, 42, 2, 22, 99, 37, 17, 67, 14, 27, 33, 7, 6, 65, 25,
odds: 99, 37, 17, 67, 27, 33, 7, 65, 25,
the odds are doubled : 198, 74, 34, 134, 54, 66, 14, 130, 50,
previous odds doubled: 198, 74, 34, 134, 54, 66, 14, 130, 50,
letter 'C'
symbol '+'
symbol '+'
number '2'
number '3'
symbol ' '
letter 'i'
letter 's'
symbol ' '
letter 'g'
letter 'r'
letter 'e'
letter 'a'
letter 't'
symbol '!'

Figure 15.4: Output of LookAtFilterAndTransform

Our result shows the range adaptor output a list of strings.

Recipe 15.6: Working with the join() and split() adaptors

Problem: It would be good to have utility functions that can split and join ranges.

Solution: There are a few functions to that from the C++20 release.

Typical tasks that a developer have: we have a bunch of items and we want to combine them into one entity, or we have one entity and we need to break it up into smaller ones. Ranges have new functions to help with that work. In the following code example, we have a container of containers of numbers, and what we want to do is combine them into a single view. The adaptor ranges::join_view() creates a single view that you can iterator over. We can see that before the call, we had 5 groups of various lengths, totaling 15. After the call, the result was 1 group with a length of 15. The nice thing about this, is that we did not copy values into a new entity; ranges::join_view() knew how to connect them into one seamless result.

The next code block, is a similar example, but might not exactly be what you want. We are trying to join a container of strings. When we look at the container, we can see that we have 3 items. Once we join them, we look (using and see that the value is 15. It seems odd that we have more items when trying to join something, but let us look at the previous example again.

Previously, we had a container of containers of numbers. This time, we have a container of strings, which is not the same number of levels, until we realize that it is processing it as a container of container of characters. That is why

we have a larger number afterwards because it is a character count. It still is usable, but when many other languages have a join function that combines a collection of strings into one string, with ours it is not what we initially guessed would have happened. The ranges::join_with_view() is different from some other languages, they create a new entity and copy the values, which is not what views are about.

```
{



{

<< << "Examples in: " << __FUNCTION__ <<

const bunchesOfNumbers { };

for = i < bunchesOfNumbers.size(); i++) {

  " + + + ", size=" + bunchesOfNumbers[i]);

  }

const

auto d =

  join, size=" + + ": joinedNumbers);

<<
```

```
words = { "C++23" };



<< "Words, size=" + + ":

for : words) << w <<

<<

const

  d =

<< "joined characters, size=" + + ":

for << s;

<< <<

const sentence = { "I like C++23" };

<< "sentence, size=" + + ": " + sentence <<

const splitter = "

// NOTE: (at least for MS VS compiler) splitter must be a separate variable,
otherwise will not split
```

```
= splitter);

<< "split words: " <<

for : splitWords) {

<< wi.end())) <<

  }

<<


const manyNumbers { 25 };

  delimited manyNumbers);

const =

const splitterNumber);

  d =

=

for {

  " + + + ", size=" +
```

```
g);

splitCounter++;

  }

}

};
```

Our next code block splits up a value. That too works; it just takes a little bit more effort than we would have hoped. We start with a string that contains some words separated with spaces, and we just call views::split() to create a result of individual words. The trick is with the result type. The result has a collection of begin and end iterators, so to get our words, we have to pass those iterators to a We just have to remember that extra work is the penalty of not duplicating any values, we are still just indirectly looking at the original string.

You can see the result of our previous code block in the following figure. We take a long container of numbers, which has a value (in this case 99) acting as a delimiter. After we call we have several groups of values.

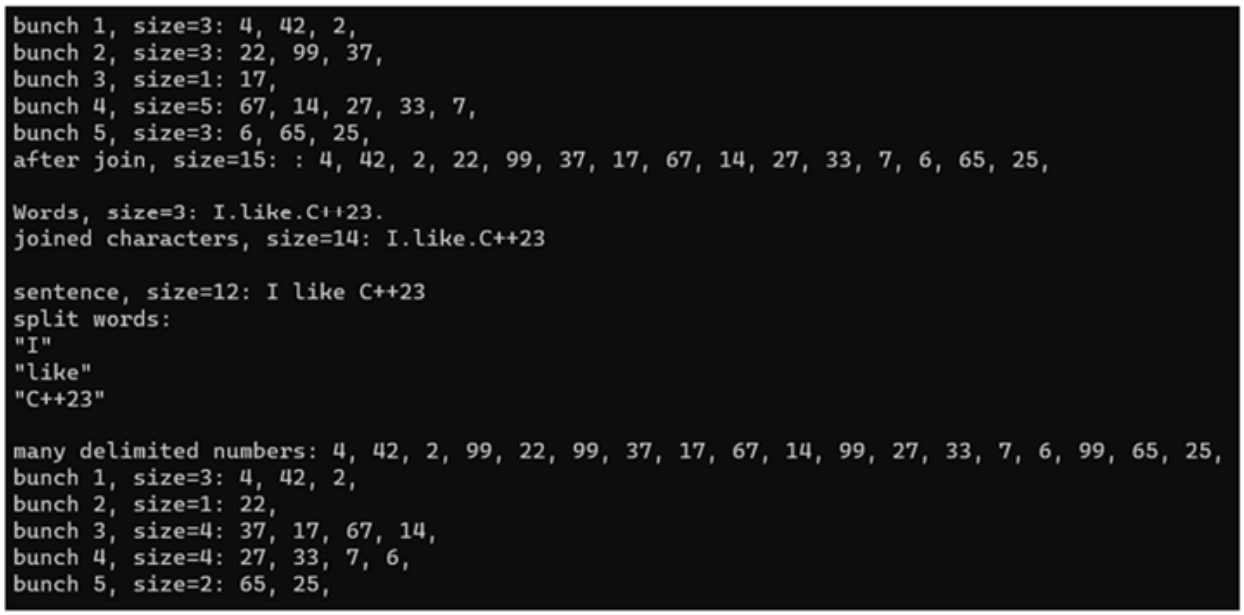
bunch 1, size=3: 4, 42, 2,
bunch 2, size=3: 22, 99, 37,
bunch 3, size=1: 17,
bunch 4, size=5: 67, 14, 27, 33, 7,
bunch 5, size=3: 6, 65, 25,
after join, size=15: : 4, 42, 2, 22, 99, 37, 17, 67, 14, 27, 33, 7, 6, 65, 25,
Words, size=3: I.like.C++23.
joined characters, size=14: I.like.C++23
sentence, size=12: I like C++23
split words:
"I"
"like"
"C++23"
many delimited numbers: 4, 42, 2, 99, 22, 99, 37, 17, 67, 14, 99, 27, 33, 7, 6, 99, 65, 25,
bunch 1, size=3: 4, 42, 2,
bunch 2, size=1: 22,
bunch 3, size=4: 37, 17, 67, 14,
bunch 4, size=4: 27, 33, 7, 6,
bunch 5, size=2: 65, 25,

Figure 15.5: Output for LookAtJoinAndSplit

It is very handy to have these adaptors; we just have to be careful how we use them, and what to expect. The other point to mention is in one case, we used and the other case we used On the surface, it may not appear to be a significant difference. The simplest answer is that lazy_split() does not look ahead for the next delimiter (hence the lazy aspect). The lazy_split() can support an input_range concept, which works with a single pass. The other split_view() supports a forward_range concept, and works in a multiple-pass fashion, but does not work with const ranges.

Recipe 15.7: Working with the keys() and values() adaptors

Problem: We would like adaptors to help us when working with maps or pairs.

Solution: C++ 20 has concepts named views::keys() and

Similar to other languages, it is often handy to focus on a particular part of a dictionary pair. We have view adaptors that can just look at the keys or the values. The following example is easy enough. We use views::keys() to get our keys and views::values() to get our values:

```
{



{

<< << "Examples in: " << __FUNCTION__ <<

const collection { };

const auto keys =

  keys);
```

```
const auto values =

  values);

<<

const microwavableFood{ "butter { };









<<

microwavableFood2{ { };









<<

const microwavableFood3{ { };
```

```
<<

}

};
```

The next code block is for the situation where it is not a map, but a collection of tuples with more than three values in each. Since we could not use the keys and values adaptors, they created another one called With this, we provide the index of the tuple value we want to look at. We were getting hungry, so we looked around to see what we could cook in the microwave. We found a few different types of food and created tuples with different attributes. Note though, although helpful, it seems unfortunate that the index is part of the template, and so we cannot have an iterator with a variable and get the values from all of the tuples.

Refer to the following figure:

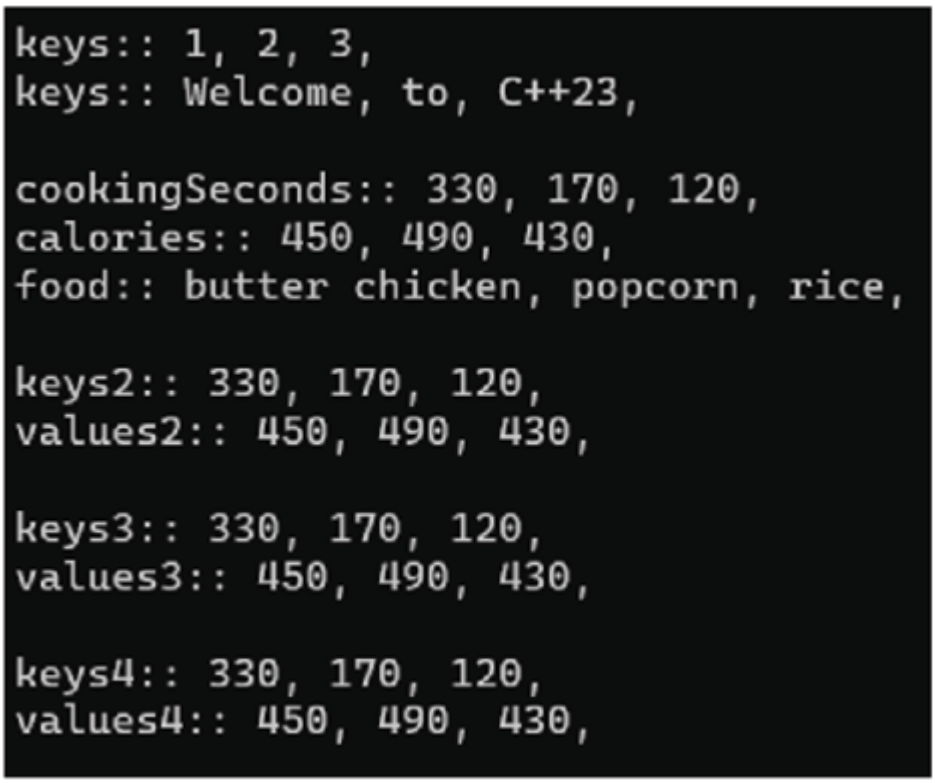
keys:: 1, 2, 3,
keys:: Welcome, to, C++23,
cookingSeconds:: 330, 170, 120,
calories:: 450, 490, 430,
food:: butter chicken, popcorn, rice,
keys2:: 330, 170, 120,
values2:: 450, 490, 430,
keys3:: 330, 170, 120,
values3:: 450, 490, 430,
keys4:: 330, 170, 120,
values4:: 450, 490, 430,

Figure 15.6: Output for LookAtKeysAndValues

The next few code blocks were more out of curiosity. First, it was to see if we had a collection of tuples with two values in them, could we use keys() and Next, we wondered if we had a collection of pairs, could we use still keys() and Lastly, if we used our original container of tuples that had three values in each, could we still use keys() and values() with that? The answer to all three questions, at least for now with the compiler we used, seemed to be yes. Since it did not seem to matter, it seems plausible that keys() were implemented with elements<0> and values() were implemented with

Recipe 15.8: Using chunk() and slide() adaptors

Problem: It would be great to have adaptors that can give us multiple subsets of our range.

Solution: While missing in the initial ranges release, C++23 has number of helpful adaptors.

Being able to easily take a large range and easily split it into manageable pieces is something extremely handy. The adaptor views::slide() probably will not be used as much as but it is a nice addition when you want to look over items multiple ways to see which window of data you want to use. If you were using the you could also just use shifting offsets, but this gives a clean call that is lazy-loaded. The following sample shows we are adjusting our window width, and outputting the various view results (via our PrintSubrange() lambda).

The views::chunk adaptor is a welcome addition. Simple enough, but great to easily break up a large range. With the pipe symbol, it looks similar to the Linux split command, being able to break a file or stream into chunks. We just specify the chunk size, and it returns a range of ranges of equal size (aside from the last one that may be smaller). You can see that views::slide() and views::chunk() are similar and you can tell them apart because slide() will always step by 1, whereas chunk() will step by the chunk size.

The adapter in the next code block is This is also handy, and similar to it is a more specialized requirement. You provide a predicate and it will return a

range of ranges base on the comparison results. Our first views::chunk_by() example uses ranges::less() that creates a new subrange when the second value is less than the first.

It is clear if you look at the following output that we start off with a subrange of {4,42} but then when get to the next value of 2, it sees that it is less so, and creates a new subrange. This continues for the rest of the input, and since the last 4 values all increase the means, the last subrange becomes {14, 27, 33, 65}. We have another example with a sentence and we tell it to create a new chunk when we see a space (‘ ‘). This is useful in specific conditions, in that if you had multiple characters to chunk by, you are not retaining what the chunked part was. The upside for something like this is that where other sentence splitters normally create copies of the values, this is doing it lately and does not allocate extra memory.

```
= r) {

<<

for = i < i++) {

if > << "

<<

}

<< "},

};
```

```
{



{

<< << "Examples in: " << __FUNCTION__ <<

const manyNumbers { 65 };


  manyNumbers);

<<

for width = width <= width++) {

const auto windows = manyNumbers |

<< "Windows of width " << width << ":

PrintSubrange);

<<

  }

<<
```

```
for width = width <= width++) {

const auto chunks = manyNumbers |

<< "Chunks of width " << width << ":

PrintSubrange);

<<

  }

<<

auto chunkBys = manyNumbers |

<< "Chunks of using < :

PrintSubrange);

<< <<

const sentence = { "I like C++23" };

auto fn = x, char y) { == ' y == ' };

= sentence |
```

```
<< "Chunks of words:

PrintSubrange);

<< <<

  manyNumbers);

const = manyNumbers |

  of 3 strideView);

<<

}

}
```

Our last code block shows another adaptor, named Where views::slide() provides a sliding range: {{{1,2,3}, {2,3,4}, {3,4,5}, {4,5,6}} including the last item, views::stride() can do big strides to return every item. Thus, if we were to do a stride of 3 on the same data, our result would be There are items after the value of 4, but they are not including since we did not find a big enough stride.

Actually, we can look at a table to see the difference between the three The size is the size of the subrange we get back, and the step is how many items you move over. Refer to Table

Table 15.1: Comparing slide, chunk, and stride

Refer to the following figure:

```
original numbers: 4, 42, 2, 99, 37, 17, 67, 14, 27, 33, 65,

Windows of width 1: {4}, {42}, {2}, {99}, {37}, {17}, {67}, {14}, {27}, {33}, {65},
Windows of width 2: {4 42}, {42 2}, {2 99}, {99 37}, {37 17}, {17 67}, {67 14}, {14
27}, {27 33}, {33 65},
Windows of width 3: {4 42 2}, {42 2 99}, {2 99 37}, {99 37 17}, {37 17 67}, {17 67 1
4}, {67 14 27}, {14 27 33}, {27 33 65},
Windows of width 4: {4 42 2 99}, {42 2 99 37}, {2 99 37 17}, {99 37 17 67}, {37 17 6
7 14}, {17 67 14 27}, {67 14 27 33}, {14 27 33 65},
Windows of width 5: {4 42 2 99 37}, {42 2 99 37 17}, {2 99 37 17 67}, {99 37 17 67 1
4}, {37 17 67 14 27}, {17 67 14 27 33}, {67 14 27 33 65},

Chunks of width 1: {4}, {42}, {2}, {99}, {37}, {17}, {67}, {14}, {27}, {33}, {65},
Chunks of width 2: {4 42}, {2 99}, {37 17}, {67 14}, {27 33}, {65},
Chunks of width 3: {4 42 2}, {99 37 17}, {67 14 27}, {33 65},
Chunks of width 4: {4 42 2 99}, {37 17 67 14}, {27 33 65},
Chunks of width 5: {4 42 2 99 37}, {17 67 14 27 33}, {65},

Chunks of using < : {4 42}, {2 99}, {37}, {17 67}, {14 27 33 65},

Chunks of words: {I}, { }, {l i k e}, { }, {C + + 2 3},

original numbers: 4, 42, 2, 99, 37, 17, 67, 14, 27, 33, 65,
stride of 3 : 4, 99, 67, 33,
```

Figure 15.7: Output from LookAtSlideAndChunk

We will take a look at another adaptor in this family, but possibly some of you will be only looking at it. At the time of writing this, not all of the compilers had not finished implementing this feature. The adaptor views::adjacent() output values would look the same as if we were using Both use a step size of 1, and the group would contain N items. The key difference is that views::slide() returns a range, but views::adjacent groups are templated tuples similar to

```
{



// ...

{



<< << "Examples in: " << __FUNCTION__ <<

const manyNumbers { 65 };

  numbers for manyNumbers);

8 };

<< "adjacent groups, using window size= " << adjacentWindowSize << " : "
<<

#ifdef __cpp_lib_ranges_zip

const = manyNumbers |
```

```
for : adjacentResult) {

<< "{" << << ", " << << ", "

   << << ", " << << ", "

   << << ", " << << ", "

   << << ", " << << "}" <<

  }

#endif

<<

const someNumbers{ 10000 };

  numbers for someNumbers);



4 };

<< "adjacent_transform groups, using window size= " <<
adjacentTransformWindowSize << " : " <<

auto AdjacentTransformFn = ints) { return (... + ints); };
```

```
#ifdef __cpp_lib_ranges_zip

const auto adjacentTransformView

#else

<< "feature is currently not working" <<

#endif

const auto adjacentResult2 | adjacentTransformView;

for : adjacentResult2) {

<< viewValue << ",

  }

<< <<

}

};
```

The first code block above uses and then you look at the item values individually. You could also make your code longer, but a bit more flexible; by using an integer loop and if your current counter is still less than the adjacent-window size, then use This loop will not work if your window size

is larger, so probably most will opt for The other code block we have shown is where you provide a predicate and it uses a group of those values in the result. Refer to the following figure:

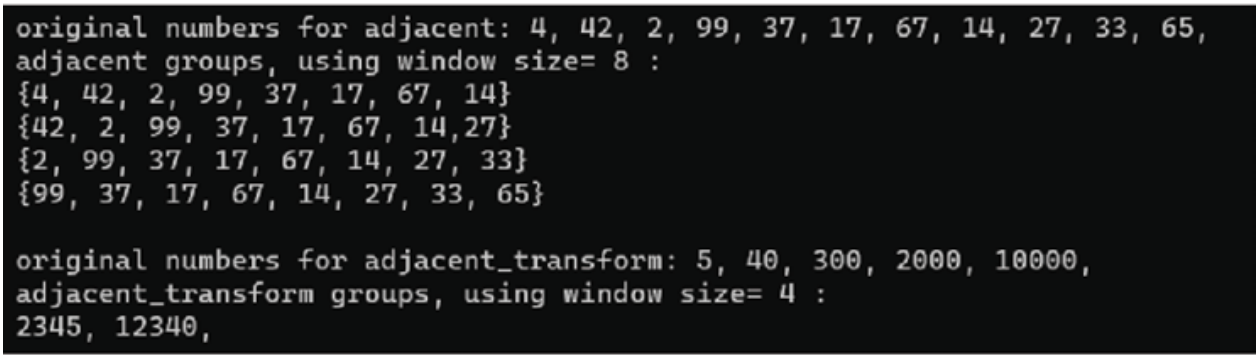
original numbers for adjacent: 4, 42, 2, 99, 37, 17, 67, 14, 27, 33, 65,
adjacent groups, using window size= 8 :
{4, 42, 2, 99, 37, 17, 67, 14}
{42, 2, 99, 37, 17, 67, 14,27}
{2, 99, 37, 17, 67, 14, 27, 33}
{99, 37, 17, 67, 14, 27, 33, 65}
original numbers for adjacent_transform: 5, 40, 300, 2000, 10000,
adjacent_transform groups, using window size= 4 :
2345, 12340,

Figure 15.8: Output from LookAtAdjactent

And once you see the output, the code becomes a lot more understandable. First, we are taking a window of 8 values from a container, and continue to do so while we slide our way along the set of numbers, as long as we have enough values to satisfy our window size. In the next code block, we are adding the sliding set of numbers together. Although you could do this in multiple steps, it gives you ideas about how these view transformations can work.

Recipe 15.9: Using zip() adaptor

Problem: It would be nice if we could splice in items from different ranges.

Solution: C++ 23 has the new views::zip() adaptor.

We have just seen different adaptors that take a range of data and break it up into multiple subranges. With we take multiple ranges and combine them into one output. In the following code example, we have a range for each meal with a few different options. The zip adaptor takes the next item of each of the ranges, and combines that into a new range in the output. Similar to some other adaptors, the result is a templated tuple that we can use std::get(elem) to get values up the the item, meaning you zipped n different ranges together.

```
{



{

<< << "Examples in: " << __FUNCTION__ <<

breakfasts{ "bacon and "porridge" };

lunches{ "chicken "club "carrot soup" };

5> dinners{ "chicken fried "salmon" };
```

```
  breakfasts);

  lunches);

  dinners);

<<



#ifdef __cpp_lib_ranges_zip

const lunches, dinners);

<< "Resulting meals" <<

for elem : mealResults) {

<< << ", "

    << << ", "

    << <<

  }

auto CombineMeal = b, auto l, auto d) { return b + ", " + l + ", " + d; };

const auto combos breakfasts, lunches, dinners);
```

```
  combos);

#else

<< "feature is currently not working" <<

#endif

}

};
```

Note that in our output, the number of combined meals is 3. This is because that was the smallest count of any of the input ranges. Despite breakfasts having 4 values, and dinners having 5 values, the lunches range only had 3 values, which means that it is the number of items in the zip output.

The other adaptor shows where you supply a predicate that takes a value from each range provided, and you can combine them into one result that gets put into the resulting range. Where views::zip() had a result showing a range of ranges, each with 3 meals (a breakfast, lunch, and dinner), views::transform() takes each of those input values and combines then into one output value that goes into the output range. In our example, we concatenated the strings. Refer to the following figure:

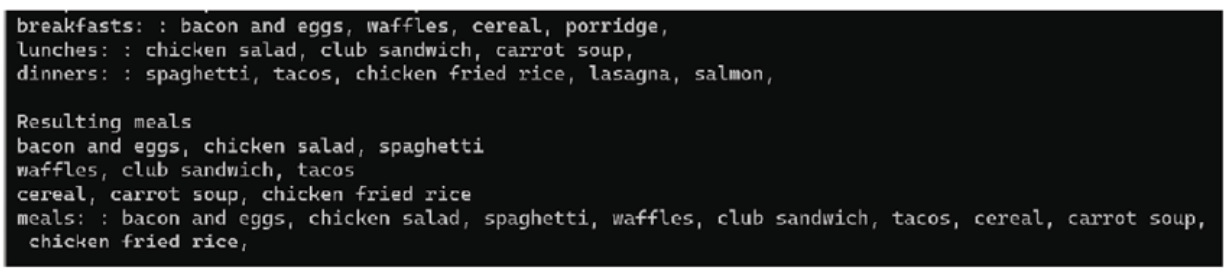
breakfasts: : bacon and eggs, waffles, cereal, porridge,
lunches: : chicken salad, club sandwich, carrot soup,
dinners: : spaghetti, tacos, chicken fried rice, lasagna, salmon,
Resulting meals
bacon and eggs, chicken salad, spaghetti
waffles, club sandwich, tacos
cereal, carrot soup, chicken fried rice
meals: : bacon and eggs, chicken salad, spaghetti, waffles, club sandwich, tacos, cereal, carrot soup,
chicken fried rice,

Figure 15.9: Result of LookAtZip

Recipe 15.10: A quick look at reverse()

Problem: We need a quick way to reverse the items in a range.

Solution: In one of the unambiguously named adaptors to understand, we have

This does not need much explanation, we have an adaptor that flips the order of the items, so the last becomes first. Keeping with the theme from the previous example, if we have a range of 3 meals, calling the adaptor views::reverse() returns them in reverse order.

```
{



{

<< << "Examples in: " << __FUNCTION__ <<

const meals { "dinner" };

  meals);

const = meals |
```

```
  Meals: backwardMeals);

const meals };

  backward Meals: moreBackwardMeals);

}

};
```

The other code block shows instead of piping a range to you can also declare a ranges::reverse_view with your range. Your variable will be set to similar reversed values, as shown:

```
meals: : breakfast, lunch, dinner,
backward Meals: : dinner, lunch, breakfast,
more backward Meals: : dinner, lunch, breakfast,
```

Figure 15.10: Look at LookAtReverse

Recipe 15.11: Looking at special adaptors

Problem: It would be good to have adaptors that alter the state, unlike most others that alter the values or sequence.

Solution: We have some new adaptors in C++20 and C++23, that help with some miscellaneous but important aspects.

We grouped these adaptors, because they did not fit into another topic nicely. They do not change the values, or group the values differently, but are useful when working with ranges.

The first adaptor we will look at in our code example is This provides a view of the provided range. It is only used in certain circumstances, but could also be useful if you are manipulating the data. The following example shows some information about a string vector. We create a and we also call The result is the same type, and we can work with it in the same manner.

Our next code block shows a simple example of This allows the view to be the owner of the data. Again, this would be a specialty case, but when required, it is handy this exists.

We can then look at This is for (mostly legacy) cases where it needs a range instead of a view. Thus, we can create a range with the same data as our view has.

The next two code blocks are again for special purposes. The first uses which turns a view into a The last uses which casts each element to an

```
{



{


<< << "Examples in: " << __FUNCTION__ <<

const message { "?" };

  message);

<< "Address of message=" << << ", size=" << <<

const message };

<< "type of ref_view is: " << <<

<< "Address of refAll=" << << ", size=" << <<

const =

<< "Address of All=" << << ", size=" << <<

<< "type of all result is: " << <<

const auto few = message |
```

```
  few);

<< "Address of take result=" << << ", size=" << <<

const few };

<< "Address of ref of take result=" << << ", size=" << <<

  of refFew);


const owner { { "great" } };

  of the

const numbers{ };

  numbers);

const numbers.begin(), };

const +

  someNumbers);

//auto cannotAdd = std::accumulate(someNumbers.begin(),
std::default_sentinel, 0);
```

```
const someNumbers };

const commonViewOfSomeNumbers.end(),

<< "Sum=" << canAdd <<

#ifdef __cpp_lib_ranges_as_const

const someNumbers };

<< "type of as_const result is: " << <<

#else



<< "feature is currently not working" <<

#endif

#ifdef __cpp_lib_ranges_as_rvalue

const someNumbers };

<< "type of as_rvalue result is: " << <<

#endif

}
```

```
};
```

Again, these are mostly for special cases, some for legacy reasons, or for times when your code needs something very explicit. Refer to the following figure:

```
message: C++23, is, great, !, Is, it, not, ?,
Address of message=000001604B8DFA60, size=8
type of ref_view is: class std::ranges::ref_view<class std::vector<class std::basic_string<char,
struct std::char_traits<char>,class std::allocator<char> >,class std::allocator<class std::basic
_string<char,struct std::char_traits<char>,class std::allocator<char> > > > >
Address of refAll=000001604B8DFA60, size=8
Address of All=000001604B8DFA60, size=8
type of all result is: class std::ranges::ref_view<class std::vector<class std::basic_string<cha
r,struct std::char_traits<char>,class std::allocator<char> >,class std::allocator<class std::bas
ic_string<char,struct std::char_traits<char>,class std::allocator<char> > > > >
few: C++23, is, great,
Address of take result=000001604B8DFA60, size=3
Address of ref of take result=000001604B8DFA60, size=3
ref of few: C++23, is, great,
owner of the message: C++23, is, great,
all: 1, 2, 3, 4, 5,
some: 3, 4, 5,
Sum=12
type of as_rvalue result is: class std::ranges::as_rvalue_view<class std::ranges::subrange<class
 std::counted_iterator<class std::_Deque_iterator<class std::_Deque_val<struct std::_Deque_simpl
e_types<int> > > >,struct std::default_sentinel_t,1> >
```

Figure 15.11: Output from LookAtSpecialAdaptors

For these special adapters, we can repeat they are for special uses. If you see them and wonder what am I going to use them for? Then you probably do not need to worry about them. There will be a few people out there who see them and yell That will be handy!

Conclusion

After this chapter, you see the power that the ranges library brings. You can start with a container, which is interpreted as a range. We can then call one or more adaptors, which results in a view. We also saw with the pipe symbol that it is really easy to pass containers (or other types) into adaptors, and we can even chain adaptors together! The ability to separate, manipulate, and join data back together is one of the most powerful features C++ has had in many releases. You can change the values, and/or what you are looking at, without using loops, or long custom functions, or lots of temporary variables. We wanted to give one last example about how powerful ranges and views can be. Although there were some hacks, and using a third-party library in spots, it is worth watch a CPPCon video on making a calendar.

Points to remember

While ranges were originally released in the C++20 release, there were some new adaptors in C++23.

Using the pipe symbol to compose multiple adaptors together is incredible.

Think about how you want to look at your data because there is probably an adaptor that will take how your data is, to how you want to use it.

Adaptors still give you the ability to transform and filter your data. This gives you extra power to easily change the values you are looking at.

Most adaptors are lazily executed, and so it does not process the value until it needs to. In addition to that, since adaptors can be chained together, you could see a nice performance boost.

There may be some legacy, or special reasons that you will have to adjust the state of your data. Luckily, there are some adaptors for those cases as well.

References

CPPreference. (n.d.). SFINAE. Retrieved from https://en.cppreference.com/w/cpp/language/sfinae

YouTube. (n.d.). RangeAndView. Retrieved from https://www.youtube.com/watch?v=YWayW5ePpkY

YouTube. (n.d.). SlideAndStride. Retrieved from https://youtu.be/-_lqZJK2vjI

YouTube. (n.d.) Calendar. Retrieved from https://www.youtube.com/watch?v=mFUXNMfaciE

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 16

Range Factories and Utilities

Introduction

We have talked a lot about ranges, from their concepts to all of the adaptors. This last chapter deals with factories that create ranges, and some other miscellany that did not fit nicely into the previous chapters on Ranges.

Structure

In this chapter, we will cover the following topics:

Recipe 16.1: Factory overview, and the ranges::iota_view factory.

Recipe 16.2: The views::repeat, and views::cartesian_product.

Recipe 16.3: Creating your own range factory.

Recipe 16.4: Formatting ranges.

Recipe 16.5: Swapping ranges.

Recipe 16.6: The utility ranges::to

Again, as an overview of the chapter topics, we will show you the functions we will run in this chapter. The code comments below also indicate when the feature was last tested, and whether it compiled and ran successfully. If you copy the code as a standalone program, you can change the void LookAtChapter16Topics() to int

```
{
```

```
// works with       when tested  clang

LookAtIota();                  // 2023.10.21  yes    no

2023.10.21  yes  no    yes

LookAtGenerator();             // 2023.10.21  no

LookAtRangeFormat();           // 2023.10.21  maybe* no    no


// * sometimes worked with clang (widberg) on godbolt, not the trunk

LookAtSwapRanges();            // 2023.10.21  yes

LookAtRangesTo();              // 2023.10.21  no    no yes

}
```

Objectives

By the end of this chapter, the reader will understand about range factories and other utilities relating to ranges. You will learn these functions are rather interesting; where you can generate an infinite series, or multiply ranges together, and then tailor your range into some clean output.

Recipe 16.1: Factory overview, and the ranges::iota_view factory

Problem: We would like to learn about the new ranges factories.

Solution: With the ranges library in C++ 20, they created a few factories, and the first one we will look at, is quite easy.

We have already talked about a couple of factories in a previous chapter, those being views::empty() and Since it is not difficult to build something that creates nothing, we will get into some better examples. Most of you have already created or used factories in your code; a function that will create a new object based on the inputs.

The views::iota() factory can be bounded or unbounded. In most cases, you would probably be generating bounded numbers, meaning you have set an upper bound. This is not an inclusive range, the bound is the value of what you want to use. If you look at the example:

```
#include

#include

#include

#include
```

```
#include

#include

#include

#include

#include

#ifdef __cpp_lib_generator

#include

#endif

#include


#ifdef __cpp_lib_print

#endif

title, auto view) {

= n) { << n << ", };

<< title << ":
```

```
for : view) {

PrintValues(i);

}

<<

}

{

<< << << __FUNCTION__ <<



auto numbers = |

numbers);

auto numbers2 = | n) n * });

numbers2);

auto almostUnlimitedNumbers = | |
```

```
more almostUnlimitedNumbers);

}
```

Yes, we included all of our normal headers, just so we did not have to mention them again. We also included a utility function to print our range.

You can see our first code block calls views::iota with an initial value of 1, and an upper (non-inclusive) bound of 4. We pass that result right into our PrintView() function, so so generating values is as easy as 1,2,3. Our next call shows that we do not have to use integers; there is no explicit restriction on the type, as long as it knows how to increment.

Our next code block shows a few variations on what you can do with First, we take a bounded iota factory and pipe it through This means that for the output from iota, it will step every 2. We can see in the following output that we get 4 even numbers. We then take some iota output and pipe it through views::transform with a predicate. In the predicate, it doubles the number, and so the end result are the same even numbers. Then the views::iota() factory generates an infinite number of values, but we need to have some mechanism for it to stop generating. One common way is to limit how many values to use, as we did by piping it to which stops after receiving 4 values. In between those calls, we also used It is fine to have multiple adaptors before the because it is processed lazily, you will not incur a performance hit. Refer to Figure

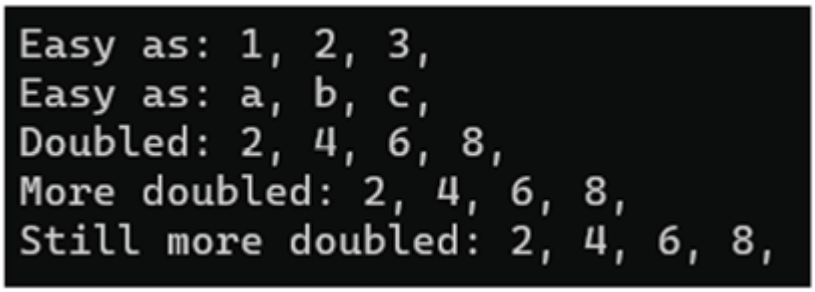

Figure 16.1: Output from LookAtIota

The differences between the newer ranges/views functions and the older functions (such as are as follows:

For you need to pre-allocate a container, whereas views::iota has a return type you can use as a view for yourself.

With you need to provide the start and end iterators, but return type has begin() and end() member functions.

The newer views::iota is lazily executed, generating values when it needs to. On the other hand, std::iota is eagerly executed, filling all the values before the call in finished.

Recipe 16.2: The views::repeat, and views::cartesian_product

Problem: We want more helpful factories to generate values.

Solution: C++23 has a couple more factories, named views::repeat and

We have seen a simple factory with Now we will start with one that is also easy, and very useful. If views::iota was as simple as 1,2,3, then views::repeat is as simple as 1,1,1! Similar to we can pass in a value and for the bounded method, we include a count. The output is a view that contains an iterator that can step through the value multiple times. In the unbounded version, we only supply the value. It assumes you will use views::take() or some other mechanism to stop it from repeating for infinity.

Our next code block has an interesting utility. Where the permutation function(s) for ranges gives you all the possibilities of one container, views::cartesian_product() gives you all the possibilities if you combine multiple containers. Our code example is looking for omelets, and we have 2 meat choices, 3 cheese choices, and 3 vegetable choices. Assuming you pick only one item of each, that will give you (2x3x3=) 18 combinations of omelets to choose from.

```
{

#ifdef __cpp_lib_ranges_repeat



do you think of this is
```

```
|

#endif

<<

auto meats = "spicy "ham" };

auto cheeses = "sharp "gruyere" , "feta" };

auto vegetables = "tomato" , "mushrooms" };

#ifdef __cpp_lib_ranges_cartesian_product

for auto & [meat, cheese, veg] : cheeses, vegetables)) {

<< "{" << meat << " & " << cheese << " & " << veg <<

<< (n++ % 4 ? "\n" : ",

}

#endif

}
```

Although on different ends of the scale in terms of complexity, you can see in the output they were both easy to generate multiple values, as shown:

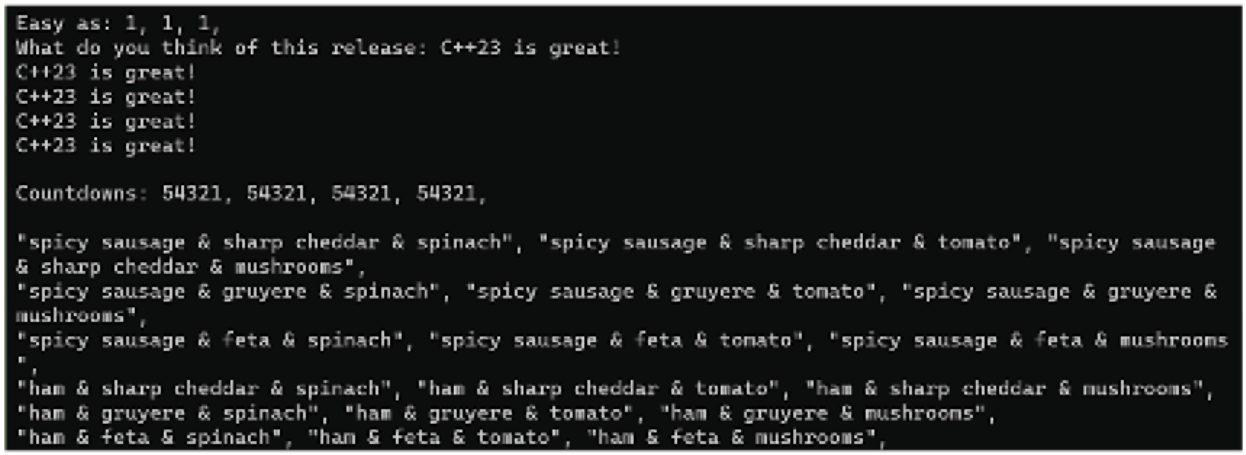
Easy as: 1, 1, 1,
What do you think of this release: C++23 is great!
C++23 is great!
C++23 is great!
C++23 is great!
C++23 is great!
Countdowns: 54321, 54321, 54321, 54321,
"spicy sausage & sharp cheddar & spinach", "spicy sausage & sharp cheddar & tomato", "spicy sausage & sharp cheddar & mushrooms",
"spicy sausage & gruyere & spinach", "spicy sausage & gruyere & tomato", "spicy sausage & gruyere & mushrooms",
"spicy sausage & feta & spinach", "spicy sausage & feta & tomato", "spicy sausage & feta & mushrooms",
"ham & sharp cheddar & spinach", "ham & sharp cheddar & tomato", "ham & sharp cheddar & mushrooms",
"ham & gruyere & spinach", "ham & gruyere & tomato", "ham & gruyere & mushrooms",
"ham & feta & spinach", "ham & feta & tomato", "ham & feta & mushrooms",

Figure 16.2: Output from LookAtRepeatAndCartesianProduct

Recipe 16.3: Creating your range factory

Problem: It would be great if we could create our own factory to generate values.

Solution: The C++ 23 release has included

It is a similar reason why they added this in addition to what already exists in the standard library. The std::generate() function works, but you need to pre-allocate your container, and your function will generate values to fill that container. With the ranges it does not have to be bounded, and does not have to pre-allocate. You can generate a value whenever it requests one, and will co_yield the result, and then wait to be called upon the next time. We learned in the previous chapter on coroutines, that it is a very clean solution for problems like this. Looking at our code example:

```
#ifdef __cpp_lib_generator

{

fibSet{ };

auto result =

auto next = fibSet.end(),
```

```
= next;

co_yield result;

}

#endif

{

#ifdef __cpp_lib_generator

FibonacciGenerator() };



Fibonacci gen |

#endif

}
```

You can see that there is not considerably less code using std::generator<> over using However, the newer version lazily executes, so it does not need to be pre-allocated or have an upper bound on it. The output would look like the following Figure

Figure 16.3: Output from LookAtGenerator

You are not limited to just integers with your generator, so this does open the possibility of following a nice Factory pattern to create objects with each request for one. (See references)

Recipe 16.4: Formatting ranges

Problem: We want an easy way out for outputting our ranges.

Solution: C++23 has accepted a feature to have range formatting.

While we have coped with std::cout for quite a while, when it comes to dynamic content, it is difficult for the compiler to know what is actually getting outputted. Often, it will not even compile since it does not understand the type, or structure of the object. A major effort in recent years has been to help the compiler (and developers) be able to output more complex types. In our case, it is ranges that we would like printed, without all the hassle of individual std::cout statements, or a lot of temporary strings while we hand-craft the output. For C++23, a proposal was accepted to have smart format specifiers that will allow formatting of ranges and tuples. The function std::format() was new in C++20, and was a welcome addition for those that had problems with We will talk more about std::format in an upcoming chapter, but they realized with the advent of ranges, that std::format should be enhanced to handle them as well.

There will be a custom formatter template that you can use, similar to the one in the fmt open-source library. (See references) Luckily though, the default/empty format specifier should suffice in many cases. The format specifier will have sections; one for the overall range, and one for the individual values. The sections will have a delimiter, the default delimiter will be a colon. If there are multiple levels, then there will be extra sections. If we look at the code example:

```
{
```

```
#ifdef __cpp_lib_format


breads { "naan" };

<< breads) <<

breadRankingsV { { }, { } };

<< breadRankingsV) <<

breadRankingsS { { }, { } };

<< breadRankingsS) <<

breadRankingsM { { }, { } };

<< breadRankingsM) <<

<<

mac{ 0x2a };

<< "The mac address is: " << mac) <<

<< "The hex value mac address is: " << mac) <<

<< "The no brackets, hex value mac address is: " << mac) <<
```

```
<< "The delimited hex value mac address is: " << mac) <<


<<

multipleValues{ };

<< "Values are: " << multipleValues) <<

<< "Hex values are: " << multipleValues) <<

<< "No bracket, filled hex values are: " << multipleValues) <<

#endif

}
```

Our first example has a vector of various breads, and all we have to do is supply the default specifier and the following output will be the strings within brackets delimited with a comma.

Our next two examples are a vector of pairs, and a set of pairs. With the default specifier, this will be outputted as a bracketed collection, with the pairs within parenthesis.

We then have a map, and the default specifier will use braces, and a colon will separate the name and value.

In our next group, we have a vector of numbers, and as expected, the output shows the comma-separated values within brackets.

We then use a format specifier of which contains:

An opening delimiter before the section for the overall output.

A delimiter for the section on the range.

A delimiter before the format of the individual values. If you recall from 02x would have the integer value shown in hexadecimal. It would also have a width of 2, and 0 padding if required.

Our next format specifier is This is similar to before, except the :n which indicates that the output of the range will not have brackets.

The last specifier for the mac address is Since it does not have enough specifiers for all the sections (the overall output, the range, and the values), it is interpreted that the lone colon is a delimiter for the output.

Our last group shows a bit more complex, a container of containers (or range of ranges is perhaps more appropriate to say). With the default specifier, it will again use brackets to group the inner range (from the vector), and it will also include brackets for the outer range (the deque).

We then try a specifier of which says the individual values are shown in hexadecimal.

The last specifier has This means:

The outer container does not need brackets around it.

The text output of the inner container will be filled to a width of 22 with the "!" character.

The individual items will be shown in hexadecimal.

Refer to Figure

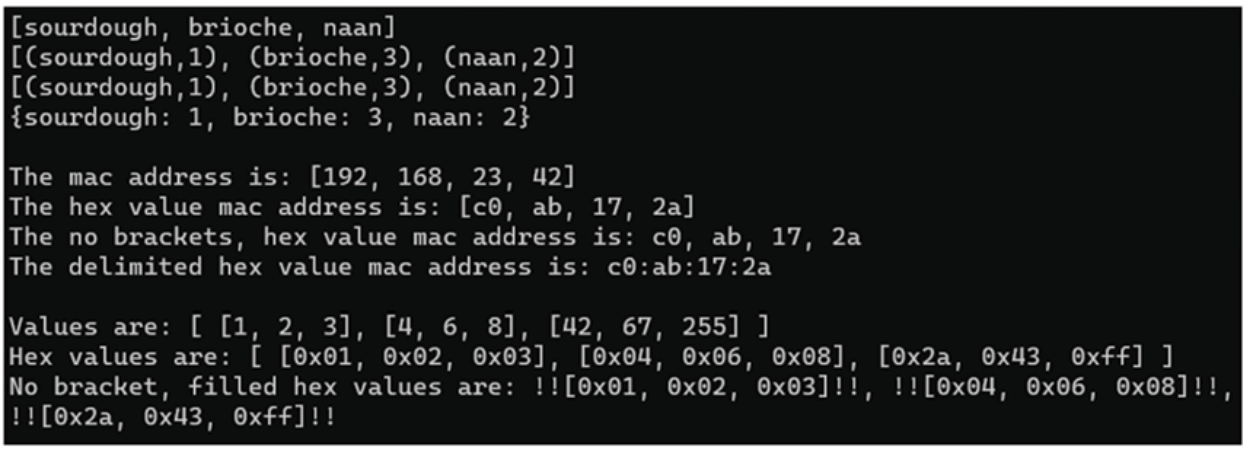
[sourdough, brioche, naan]
[(sourdough,1), (brioche,3), (naan,2)]
[(sourdough,1), (brioche,3), (naan,2)]
{sourdough: 1, brioche: 3, naan: 2}
The mac address is: [192, 168, 23, 42]
The hex value mac address is: [c0, ab, 17, 2a]
The no brackets, hex value mac address is: c0, ab, 17, 2a
The delimited hex value mac address is: c0:ab:17:2a
Values are: [[1, 2, 3], [4, 6, 8], [42, 67, 255]]
Hex values are: [[0x01, 0x02, 0x03], [0x04, 0x06, 0x08], [0x2a, 0x43, 0xff]]
No bracket, filled hex values are: !![0x01, 0x02, 0x03]!!, !![0x04, 0x06, 0x08]!!,
!![0x2a, 0x43, 0xff]!!

Figure 16.4: Output of LookAtRangeFormat

There will be more format specifications, and features for std::format() but this not only gives you the ability to print out your range without custom coding, but it does give you a fair bit of flexibility on how you display your range.

If you have more specific requirements for displaying or serializing your range, then you have some options:

Create a utility to individually output your range values to a stream.

If your range is part of a class, overload operator << so you can output your range.

Use a although that does not seem to have been fully implemented at the time of writing by the major compiler makers.

One quick mention; since we are talking about outputting our range a stream, we should also mention inputting from a stream. C++20 has ranges::stream_view<> that you can read the stream as a view.

Recipe 16.5: Swapping ranges

Problem: We want a simple way to swap ranges.

Solution: C++ 20 has a couple of new concepts, as well as the ranges::swap() function.

To complete the list, there are a couple of concepts that we did not talk about in the chapters on Concepts earlier. Technically, the concept std::swappable states that objects of type T can be swapped. It verifies that if you had objects t1 and the values could be exchanged. There is also std::swappable_with that takes 2 arguments. This concept specifies that you could swap objects t1 and

The function std::ranges::swap() takes two ranges and swaps the values. If the types are not compatible, or the counts are not compatible for the given type, then you will get a compiler error.

Before we get too far, we should clarify between a number of functions:

This function, which has been around for several releases, swaps two objects. If it sees that the two objects are arrays, then it will (effectively)

In the C++20 release, you provide the swaps the values between range1 and to the end for specified range1.

New in C++20, if the range types are compatible, it will swap all the values. It will even swap if the counts of the ranges are not equal. You do not have to specify anything else; it will use the range's beginning iterator and sentinel.

std::ranges::swap_ranges : Also new in C++20, you supply the beginning iterator and sentinel for both ranges. It will calculate the shorter distance, and swap values between the two ranges for that distance.

If you look at the following code example:

```
{

int n;

};

{

numbersOdd{ 99 };

numbersEven{ 88 };

numbersOdd);

numbersEven);

numbersEven);
```

```
were odd numbersOdd);

were even numbersEven);

numbersOdd.end(), numbersEven.begin());

values after numbersOdd);

values after numbersEven);

<<

easyNumbers{ 3 };

values easyNumbers);



easyNumbers);

were odd numbersOdd);

were easy easyNumbers);

<<

array1{ };
```

```
array2{ };

array3{ };

array1);

array2);

array3);

array2);

after 1-2 array1);

after 1-2 array2);

// cannot swap different sizes for all container types

// std::ranges::swap(array1, array3);

array3);

after 1-3 array1);

after 1-3 array3);

<<
```

```
hardNumbers{ 10 };

// cannot swap, since one instance is const

// std::ranges::swap(numbersOdd, hardNumbers);

// std::ranges::swap_ranges(numbersOdd, hardNumbers);

// std::swap_ranges(numbersOdd.begin(), numbersOdd.end(),
hardNumbers.begin());

otherNumbers{ 3 };



// will not compile, since cannot swap between these different containers

// std::ranges::swap(numbersOdd, otherNumbers);

numbersEven);

otherNumbers);

auto otherNumbersSwapResult = otherNumbers);

after numbersEven);

after otherNumbers);
```

```
103 };

};

// works on some compilers

// std::ranges::swap(digits, numbersHeld);

// std::ranges::swap_ranges(numbersHeld, digits);

// std::swap_ranges(numbersHeld.begin(), numbersHeld.end(),
digits.begin());

}
```

Our first case is simple enough to use since they have the same type and same count. We then swap the values back using

In the next example with the second vector contains fewer values than the first vector. In this case, we are still allowed to perform the swap, and all values are swapped.

We repeat those attempts, but this time with arrays. If the counts are the same, then it is allowed. This time, when trying to swap arrays of different sizes, we get a compiler error for We are allowed to perform and as you can see in the output below, it only swaps 2 values in the arrays, since that was the smaller count. Refer to Figure

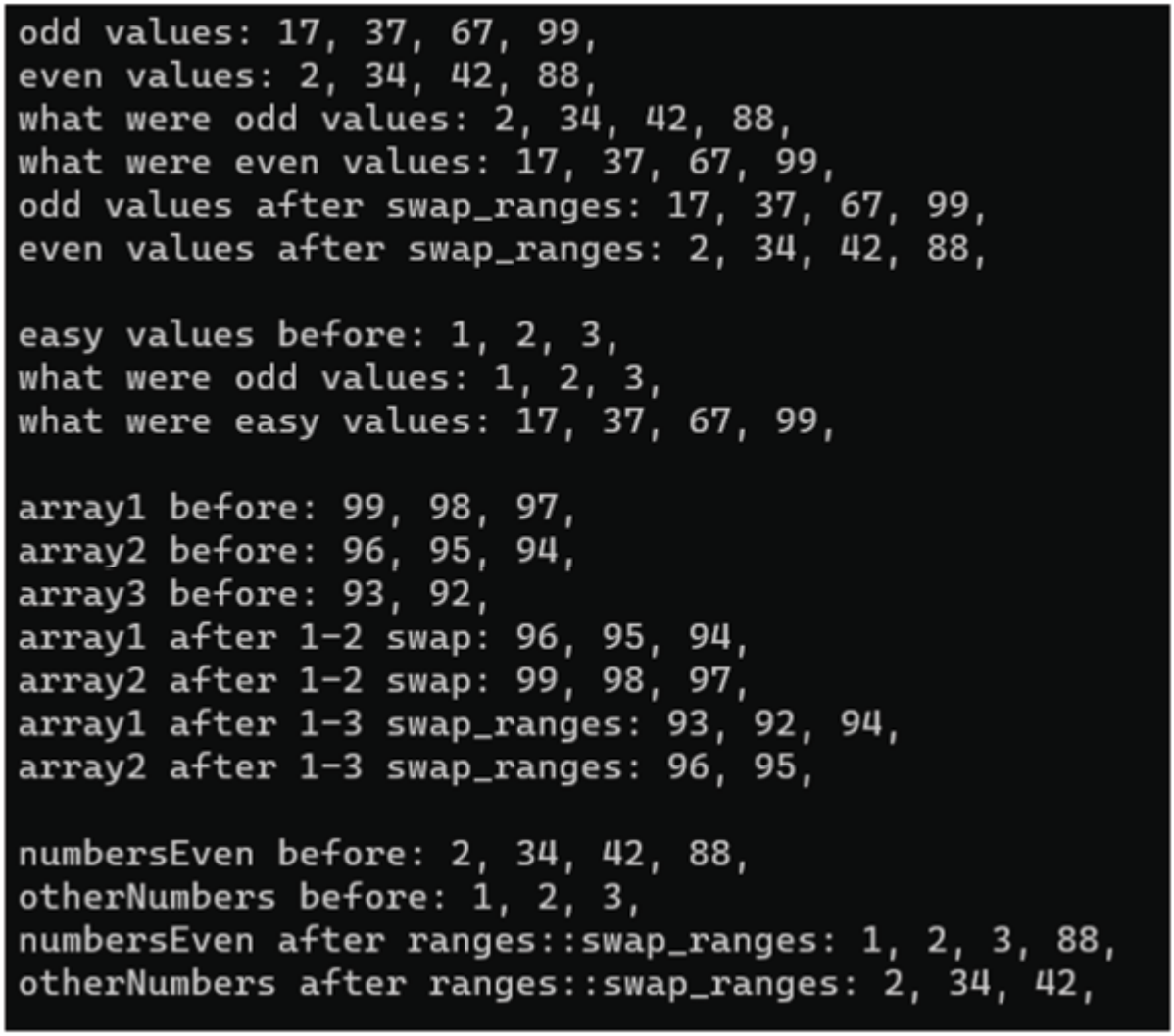
odd values: 17, 37, 67, 99,
even values: 2, 34, 42, 88,
what were odd values: 2, 34, 42, 88,
what were even values: 17, 37, 67, 99,
odd values after swap_ranges: 17, 37, 67, 99,
even values after swap_ranges: 2, 34, 42, 88,
easy values before: 1, 2, 3,
what were odd values: 1, 2, 3,
what were easy values: 17, 37, 67, 99,
array1 before: 99, 98, 97,
array2 before: 96, 95, 94,
array3 before: 93, 92,
array1 after 1-2 swap: 96, 95, 94,
array2 after 1-2 swap: 99, 98, 97,
array1 after 1-3 swap_ranges: 93, 92, 94,
array2 after 1-3 swap_ranges: 96, 95,
numbersEven before: 2, 34, 42, 88,
otherNumbers before: 1, 2, 3,
numbersEven after ranges::swap_ranges: 1, 2, 3, 88,
otherNumbers after ranges::swap_ranges: 2, 34, 42,

Figure 16.5: Output for LookAtSwapRanges

Next, we try to swap two ranges of the same type, but since hardNumbers is constant, all of the various functions fail to compile.

The code block after that wants to swap values between a vector and a deque. The function ranges::swap does not compile, but we are allowed to call Notice these ranges had different counts, so it swapped 3 values, since that was the smaller count.

The last example, where we try to swap an integer array with an array of structs that just have an integer.

Recipe 16.6: The utility ranges::to

Problem: We want a mechanism to create a new container from the contents of a range.

Solution: In the C++23 release, we have

We have seen how powerful and vast all of the new ranges functionality is. We can generate values, filter values, transform values, and much more. We can do all this, while working in the context of ranges and views. We do not have to deal with the source containers. There are occasions though, where we will need to work with a container, and so they created a function named ranges::to<>() that creates a container for you. If you look at our example:

```
{

auto LessThan99 = n) { return n < };

auto Doubler = n) { return n * };

source{ 99 };

source);

auto result = source | |

  |
```

```
after source);

of result);

<< "Result type= " << <<

}
```

We start with a deque; we pipe it in the views::filter range adaptor, and then we send it to transform. We want to capture the result in a new container, and so we pipe that to As you can see, in the output, we filtered out the 99 value, and everything else is doubled. The return value of ranges::to is a vector as we specified.

```
source: 17, 37, 67, 99,
source after work: 17, 37, 67, 99,
result of work: 34, 74, 134,
Result type= class std::vector<int,class std::allocator<int> >
```

Figure 16.6: Output of LookAtRangesTo

Conclusion

We have covered a tremendous amount of functionality in this chapter on Ranges. While this chapter was shorter than some of the others, it was certainly as useful. We learned about generators and a couple of the pre-built ones. Most likely, you will enjoy using iota versus having to create a for loop with a counter. We also looked at how to make a cartesian product of ranges, which for some of you will enjoy working with that simplicity. We then dove into the ways we can format a range, which is again a way to reduce your code from possibly an entire function to simple specifiers. Next, we looked at the various new functions on how to swap ranges. We can swap entire ranges, or sometimes a partial swap if the range counts are not the same. Lastly, we looked a an easy way to convert the view that you have been working on to a container that can be used elsewhere.

Points to remember

Foremost, it is important to check if the range features you are wanting are available with your compiler.

Using or your own generator, can help reduce loops, and create more functional code.

It will be way easier to format ranges now, but check all the various specifiers to see which would work best for you.

Swapping ranges is now very easy, but you will need to be aware if your container types are what you are allowed to swap. Moreover, check if the swap function you are using is swapping the entire range, or just up to the smaller range distance.

The ranges::to function is great to create a new container of values. Remember that if you are only using range adaptors, then your original container has not been modified, but it would be modified when you use range algorithms.

References

Wikipedia. (n.d.).

Github. (n.d.)

Join our book’s Discord space

Join the book’s Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 17

New Features for Containers

Introduction

Containers have been a big part of C++ for quite a while. In C++20, there were no huge changes related to containers; that spotlight was on ranges/spans/views that we have already talked about. However, there are a few new methods that you will appreciate using. An important new type is span, and we will discuss a bit more about that. We can also skew the counts and say that C++23 has the second-greatest number of new containers introduced. Although the flat containers are similar, nothing will compare to the C++11 release. The new containers are useful, and so we will be looking at those. There are some minor changes; such as std::reduce (from C++17) and std::transform_reduce now have the ability to be which will certainly help in some situations, and so on.

Structure

In this chapter, we will cover the following topics:

Recipe 17.1: Contains

Recipe 17.2: Looking at std::span

Recipe 17.3: std::counted_iterator

Recipe 17.4: std::is_bounded_array and std::is_unbounded_array

Recipe 17.5: std::to_array converts to std:array

Recipe 17.6: std::erase and std::erase_if

Recipe 17.7: std::flat_map and std::flat_set

Recipe 17.8: Iterators pair constructors for stack and queue

Recipe 17.9: Allow default arguments for pair's forwarding constructors

Recipe 17.10: The push_range() function for queue, stack, and priority_queue

Recipe 17.11: What is std::mdspan

As an overview of the new features we will be looking at, you can see a lot has been fully implemented across the major compilers. Unfortunately, there are still a number of features that none of the compilers have implemented.

```
{

// works with compiler?   when tested  clang

LookAtContains();      // 2023.10.21  yes

LookAtFlatMapSet();    // 2023.10.21  no    no no

2023.10.21  yes


LookAtBounded();       // 2023.10.21  yes

LookAtToArray();       // 2023.10.21  yes

LookAtErase();         // 2023.10.21  yes

2023.10.21  yes

2023.10.21  no    no no
```

```
LookAtPushRange();      // 2023.10.21  no    no yes

LookAtMdspan();         // 2023.10.21  no    no no

}
```

Objectives

By the end of this chapter, the reader will understand the new classes, along with new functionality for existing containers. Containers have been around for a long time, but you will learn there are some simple new functions that make them more useful. You will also learn of some new types that you might find handy. You will also learn of some new language features for constructors and arguments.

Recipe 17.1: Contains

Problem: We want an easier way of checking if our container contains a value.

Solution: A contains() function has been added to some of the containers.

While it was previously possible to check if a value existed in a container, it looks better (and sometimes runs faster) when you can do it with less code. For the std::set and std::map containers, it was always possible to search for a value in a loop, or compare your find() search result with In C++20, they reduced your typing and so you can just call contains() which gives you a binary result. The following code example (after the usual container headers) in Table shows how you can call

Table 17.1: contains() usage

We cannot perform contains() on a stack or queue, which makes sense. We cannot iterate through those values without altering the container. We also mentioned that you could pass a set or a map to but that has no guarantee and it will be as efficient as calling contains() on the object itself.

```
#include "Chapter17.h"

#include

#include

#include

#include

#include

#include

#include

#include

#include

#include

#include

#include
```

```
{

v{ 67 };

l{ 67 };

a{ 67 };

d{ 67 };

q{d};

st{d};

se{ 67 };

m{ };

message{ "C++23 is great!" };

ms{ 4 };

mm{ };



<< "vector can use ranges::contains=" << << <<
```

```
<< "list can use ranges::contains=" << << <<

<< "array can use ranges::contains=" << << <<

<< "deque can use ranges::contains=" << << <<

//std::cout << "queue cannot use ranges::contains=" << std::boolalpha <<
std::ranges::contains(q, 42) << std::endl;

//std::cout << "stack cannot use ranges::contains=" << std::boolalpha <<
std::ranges::contains(st, 42) << std::endl;

<< "set contains value=" << << <<

<< "set can use also ranges::contains=" << << <<

<< "map contains value=" << << <<

<< "multiset contains value=" << << <<

<< "multimap contains value=" << << <<


<< "string contains value=" << <<

}
```

```
{

LookAtContains();

}
```

The other positive aspect of this example, is that we called contains() on a string. Since you can think of string and string_view as containers of characters, once you migrate to C++23, you can easily look for values with contains() instead of the find() method, as shown in Figure

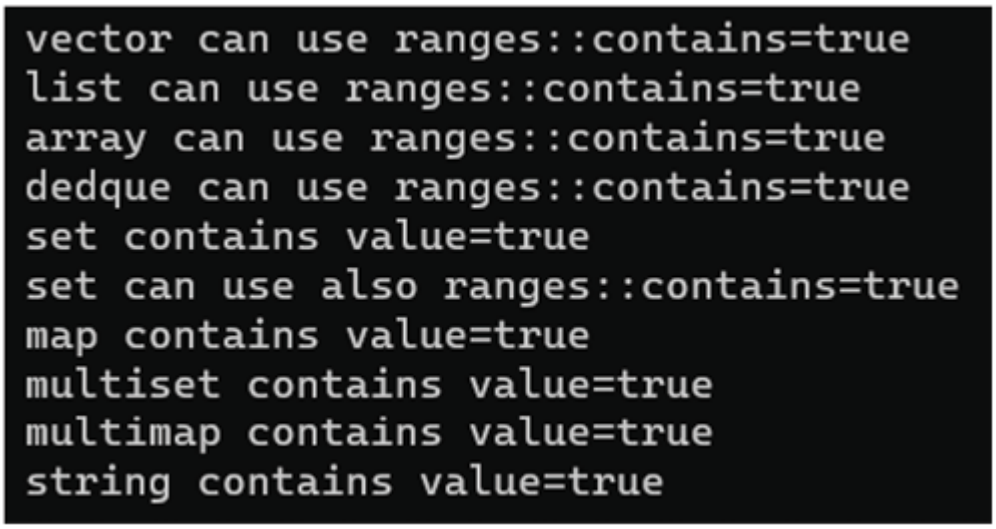

Figure 17.1: Output from LookAtContains

Recipe 17.2: Looking at std::span

We want more details about

Solution: The span class should catch on in use, and there are lots of ways to use it.

We have already talked a lot about std::span in Chapter Introduction to Ranges and when we first started talking about ranges. Since ranges relate to containers, and spans relate to ranges, they mesh together nicely.

We have discussed the span already, but there are a few more details you should know:

The assignment operator when it copies the assignment, is just copying the pointer and the size.

It was originally planned for the comparison operator to be shallow as well. Since it has more reference semantics, and not owning the data, it makes more sense to use comparison on a simplistic level. It was too confusing since we say std::span is similar to but string_view does do a deep comparison. The end decision was for std::span to not have a comparison operator. It was partial because span, unlike can modify data that it is looking at, so it seemed clearer to state it was rather than being inconsistent.

There was a requirement in C++23 to make std::span (along with trivial copyable. What it means is that you can copy your object into an array of bytes or characters, and if you copy that array back, it should retain its original value. You can almost think of it as being serializable, although it is not the same. There is an older check from C++11 called std::is_trivially_copyable<> that will check on your type.

Recipe 17.3: std::counted_iterator

Problem: We want an iterator that also knows how far it is until the end of the range.

Solution: In the C++20, they created

Iterators are handy in many situations, but there are times when you would like to know how many items are between your current iterator spot and the end of the container that you are looking at. That is exactly what std::counted_iterator can do. With ranges::subrange, which has a finite ending, you can calculate the distance to its end. Similarly, when the counted_iterator is instantiated, you will know how many items it has. This allows you to always know how many items are in it (or how many items before it ends). If you look at the code example, we added a header to the ones we had before, and a utility function to print out our range:

```
// … add to previous headers

#include

title, span) {

<< title <<

for {
```

```
<< n << ",

}

<<

}

{

manyNumbers = { 65 };


numberCounter{ manyNumbers.begin() + 4 };

otherNumberCounter{ numberCounter };

yetAnotherNumberCounter{ numberCounter };

manyNumbers);

<< "Original count=" << manyNumbers.size() <<

<< "counted_iterator count=" << numberCounter.count() <<

for (; numberCounter.count() != ++numberCounter) {
```

```
<< "Current value=" << *numberCounter << ", distance to end=" <<
numberCounter.count() <<

}

sub{ otherNumberCounter.base(), manyNumbers.end() };

at sub);

<<

for (; otherNumberCounter != otherNumberCounter++) {

<< "Other current value=" << *otherNumberCounter << ", distance to
end=" << otherNumberCounter.count() <<

}

<<

for n = n < yetAnotherNumberCounter.count(); n++) {



<< "yet Other current value [" << n << "]=" <<
yetAnotherNumberCounter[n] <<

}
```

```
}
```

We define the counted_iterator with a beginning and a count. This allows the iterator to know how many items are left, which it does with the count() function. We also create a couple of extra variables of the same type, by passing in the counted_iterator that we created, and so it uses those same properties. After we print out the entire container of values, we then print out the values using our counter_iterator. Just like its relative, we can dereference to get its value. We also use its added feature by saying how many items are left.

In the next code block, we show another member function named which provides the iterator that it was based upon. When we print out those values, it starts at the same value, but iterators to the end. That is because when we defined the subrange, we provided the end of the container, not the end of the The std::counted_iterator does not have an end() function, nor could we use If we wanted to, we could have easily supplied the subrange's end as which would specify the iterator location that is just past the last item we want to include.

In the next step, there are two aspects to consider:

First, we are using the same variable. We just used the otherNumberCounter variable when we created the subrange. The subrange has its iterator, and so we did not alter what otherNumberCounter is pointing to. Thus, we can loop through with that variable.

We are comparing our iterator to We have used std::default_sentinel earlier in the chapter, but it still does look a bit odd. The std::default_sentinel is an empty class used to denote the end of a range (or container). It can be used in places where you know where the end of the range is, normal for a subrange, or in this case, a

Lastly, we are using another variable that we defined earlier, so the starting point has not changed. This time, we are looping through getting the item with its index, which of course you can do if you know the count of items. The one thing that we are missing, since we are not using the iterator, is that we cannot use it to print out how many items are left. (We know the total count, so we could calculate it, but we do not have the iterator to provide it automatically). Refer to Figure

```
Original numbers
4, 42, 2, 99, 37, 17, 67, 14, 27, 33, 65,
Original count=11
counted_iterator count=4
Current value=99, distance to end=4
Current value=37, distance to end=3
Current value=17, distance to end=2
Current value=67, distance to end=1
Looking at numbers
99, 37, 17, 67, 14, 27, 33, 65,

Other current value=99, distance to end=4
Other current value=37, distance to end=3
Other current value=17, distance to end=2
Other current value=67, distance to end=1

yet Other current value [0]=99
yet Other current value [1]=37
yet Other current value [2]=17
yet Other current value [3]=67
```

Figure 17.2: Output from LookAtCountedIterator

Recipe 17.4: std::is_bounded_array and std::is_unbounded_array

Problem: We want an easy way of checking if we are using a bounded array.

Solution: C++20 has new utility functions named and

With all of the new concepts and other modern features, the C++ standard is providing ways for you to do more runtime type checking. To ensure that the type you are using is a bounded array, they provided a new function called To clarify, that is a C type array with a bounded size. If you look at the following code sample, we have set up a quick templated function to check if it is a bounded array, and if it is an unbounded array.

```
// … add to previous headers

#include

T>

title) {

=

<< << title << (isBounded ? " is bounded" : " is not

=
```

```
<< ", and is " << (isUnbounded ? "unbounded" : "not <<

}

{

}
```

This results in the following output:

```
int is not bounded, and is not unbounded
int[] is not bounded, and is unbounded
int[3] is bounded, and is not unbounded
int[][3] is not bounded, and is unbounded
int[2][3] is bounded, and is not unbounded
std::array<int, 3> is not bounded, and is not unbounded
std::vector<int> is not bounded, and is not unbounded
std::counted_iterator<std::vector<int>::iterator> is not bounded, and is not unbounded
```

Figure 17.3: Output from LookAtBounded

The first is fairly clear, although it is odd to see something with both tests showing negative. A simple int is not bounded nor unbounded, because it is not an array. The only tests that show they are bounded are where the C style arrays have all dimensions specified. The arrays that would use indices but no value specified, then those tested as unbounded. Also note that a std::array<> did not test as bounded, nor unbounded.

Recipe 17.5: std::to_array converts to std:array

Problem: We want an easy way of converting a C style array to a

Solution: In the C++20 release, there was a utility function named

Since we were just talking about arrays and we can talk about a new utility function. There are times when a simple C style array is handy, and then there are times when you want a std::array object. The newer, templated class:

Allows for the typical container semantics (such as copy)

Does not decay to a pointer as the old [] array does.

Gives a modern feel, without any performance loss.

Can do everything its predecessor did, except easily handle multiple dimensions.

In the cases where you need (or want) a instead of manually creating it, you can do it with one call. Our code example uses a few more headers such as:

```
// ... add to previous headers

#include

#include
```

```
#include

#include

{

auto c1 = { };

auto c3 = { 5.6 };

m4 = {

auto s5 = "Hello


int =

int = { };

// c1 is a std::initializer_list and cannot be used in this case

// auto a1 = std::to_array(c1);

auto a2 = });

<< "Resulting type of a2: " << <<

auto a3 = 5.6 });
```

```
<< "Resulting type of a3: " << <<

// cannot convert since m4 is a map

// auto a4 = std::to_array(m4);

auto a4 = });

<< "Resulting type of a4: " << <<

// cannot convert a const char *

// auto a5 = std::to_array(s5);

auto a6 =

<< "Resulting type of a6: " << <<

auto a7 =

<< "Resulting type of a7: " << <<

// compilter error for multidimensional arrays

// auto a8 = std::to_array(c8);

}
```

We do not start off well, because we found if you use auto to define your array, then the compiler defines it as a which does not (yet) work with We see it works if you pass it with the array defined as a parameter, but that probably is not too likely. We have a similar problem if we pass it a map, but it will accept it if you declare the values right in the parameter. Again, a compiler error if we define a string with auto, which assumes it is a const char We do see it work if we explicitly declare our integer array, but we also see that it does not work with multidimensional arrays. Refer to Figure

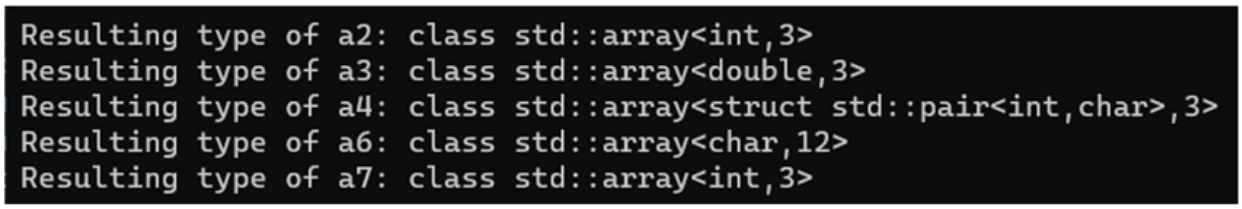

Figure 17.4: Output of LookAtToArray

From the preceding output, we can see it is a handy tool, as long as you are in the right circumstances to use it.

Recipe 17.6: std::erase and std::erase_if

Problem: We want an easier way of removing data from our container.

Solution: There were new std::erase() and std::erase_if() functions added in C++20.

Many of you have probably used the erase() member function that is provided with containers such as and There is also a similar erase_after() function for These are iterator-based, which means that first you have to search for the value, then see if the resulting iterator is valid, and only then, you can finally erase the value. While not a painful amount of code to write, people thought it could be done more easily. That is why they created the value-based std::erase() function, where you provide a value to remove if it exists. The return value is the number of items erased, which helps verify your action.

For the times when you are not looking for a particular value, but more of a condition on when to erase something, there is This takes a predicate and will erase the current item if the predicate returns true.

We added another header to use our code example:

```
// ... add to previous headers

#include
```

```
{

originalNumbers = { 65 };

originalNumbers);

manyNumbers{ originalNumbers };



auto gone = manyNumbers.erase(manyNumbers.begin() +
manyNumbers.begin() +

vector manyNumbers);

<<

manyNumbers2{ originalNumbers };

manyNumbers2);

auto result2 =

<< "# of 42 erased=" << result2 <<

auto result3 =

<< "# of 999 erased=" << result3 <<
```

```
manyNumbers2);

<<

auto EndsWith7 = n) { return n % 10 == };

manyNumbers3{ originalNumbers };

manyNumbers3);

auto result4 = EndsWith7);

manyNumbers3);

<< "# of *7 erased=" << result4 <<

}
```

First, the example uses the old member function and iterators to delete some items. Then with a new we show that it erased 1 item with the value 42, and 0 items with the value 999. Lastly, we erase any items that end with a 7, of which there were 3 of them, as we can see in the following output:

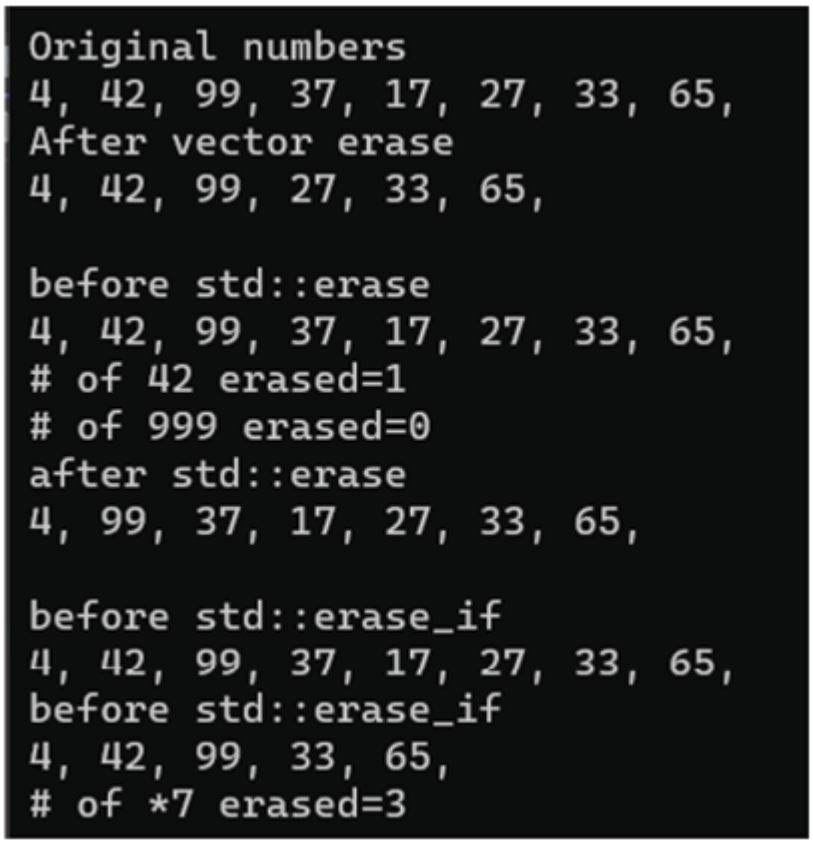
Original numbers
4, 42, 99, 37, 17, 27, 33, 65,
After vector erase
4, 42, 99, 27, 33, 65,
before std::erase
4, 42, 99, 37, 17, 27, 33, 65,
# of 42 erased=1
# of 999 erased=0
after std::erase
4, 99, 37, 17, 27, 33, 65,
before std::erase_if
4, 42, 99, 37, 17, 27, 33, 65,
before std::erase_if
4, 42, 99, 33, 65,
of *7 erased=3

Figure 17.5: Output from LookAtErase

The std::erase() function works with the same types that have the erase member function (vector, deque, list, set). However, std::map does not work with which makes sense because you probably do not have the key-value pair to provide it with. The type std::map can use the std::erase_if() function.

Recipe 17.7: std::flat_map and std::flat_set

Problem: We want containers that provide fast key retrieval.

Solution: C++23 has and

We said at the beginning of the chapter that C++20 did not have many changes in terms of the basic containers. An arguable, but understandable statement, since we just spent the last several chapters on range functionality that wraps around containers. We can also argue that C++23 was the second biggest release ever in terms of new container classes. Most of these new classes are similar, relating to existing map and set types. Everybody uses the set and map; they contain fast lookups via the key value. We often think of std::map as a collection or Most know of, if not used, multiset and They are similar containers, but do not have the restriction of a unique key.

Now in C++23, we will also have something called The class (or technically they are container adaptors) have similar functionality, but built differently. The keys and values are actually stored in separate containers (normally a std::vector or By doing this, it allows the keys to be sorted, which means faster retrievals. There is no performance gain when inserting, so it is targeting in cases where you perform a lot more lookups versus insertions.

After std::flat_map is which is what you expect, based on what we just talked about. The container std::set is a collection of values; you can add a value, and check if a value is there, but std::set was not designed for speedy checking if a value existed in the set. The new container is basically the keys portion of It was built for faster lookups, so if you are doing more lookups than additions, then std::flat_map would be a better choice over the older

With the next two containers: std::flat_multimap and we have the same characteristics as the two other new containers and except you are allowed to have the same key multiple times. The main points for both of the multi-containers are for faster lookups, along with the multiple keys of the same value.

Since compilers have not implemented these containers yet, the full details are still not confirmed. Some of the planned functionality is:

Most likely the flat_map will have a has() function. Here, contains() returns a Boolean, and the has() will return a pointer (or perhaps an iterator) to the value in the container. The paper that was submitted also said that if the value was not found, then it would return null. That seems to contradict the normal return type in cases that it was not found, that it would be an end() value.

There will be an overload for insert to signify that the container does not have to sort as soon as the value was added. This might be up to the characteristics of the individual compiler, but if you are adding values to the container and want to delay the sort, then perhaps there should also be a way of triggering the sort. The idea is good, but we are waiting on the implementation fine details.

There might be a binary_find() free function, although it seems too similar to the std::binary_search() function added in C++20.

We mentioned that it was actually a container adapter, versus a container. The std::stack and std::queue are also adaptors, although we do not think that much about their underlying storage. With std::flat_set and they are proposing the underlying storage to be clearly stated. The declaration might be similar to std::flat_set> which not only tells you that it stores integers, but it is using/adapting to a vector. The proposal also said that there would be typedef aliases for them, and actually the preferred declaration would use the typedef of along with We can also mention the proposal suggests a couple of public data members that refer to the underlying container. The first data member is simply named "container", which would obviously provide access to the container. The other is a member named which would allow you to access an iterator to the underlying container.

Recipe 17.8: Iterators pair constructors for stack and queue

Problem: We want a quick way of adding to a container via iterators.

Solution: C++23 added the ability to construct a container based on a iterator pair.

We already could create a std::stack or std::queue based on the values from another container (such as a vector or What C++23 brings us is the ability to create a stack or queue with a subset of values, by providing begin() and end() iterators. We have a simple example:

```
{

dNumbers{ 65 };

auto count = dNumbers.size();

for = i < count; i++) {

<< "number[" << i << "]=" << dNumbers[i]

 << && * " : " <<

}
```

```
stNumbers1{ dNumbers };

<< "Stack1 size=" << stNumbers1.size() <<

stNumbers2{ };

<< "Stack2 size=" << stNumbers2.size() <<


qNumbers3{ dNumbers.begin() + dNumbers.end() - 3 };

<< "Queue3 size=" << qNumbers3.size() <<

}
```

Knowing we want a subset, starting at the from the start, and ignoring the last 3, we print out the original numbers, with an asterisk on what we will include. We then create a stack and a queue with the subset. The following output shows the container size with 4 values, which matches our expected count:

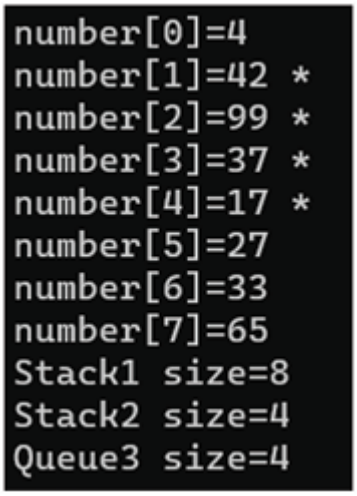

Figure 17.6: Output for LookAtPairConstructors

Recipe 17.9: Allow default arguments for pair's forwarding constructors

Problem: We want an easy way of adding a default argument.

Solution: C++23 has a plan to allow default arguments.

This is another feature that is not groundbreaking, but a nice one to have, that will clean up some code. This is a scenario where it might be easier to simply look at the code and then talk about it.

```
{

m1;

#ifdef __cpp_lib_pair_ctor_default_tmpl_args

{});

#else


#endif

}
```

Currently, there are situations where you need to instantiate something to pass it as a parameter. The preceding code has to create a new to create the pair to add to the map. Although not that painful to do, it does mean extra code that the developer has to see, understand, and maintain. The idea is, since it is empty anyway, to have a default initializer that will create the vector, that can be passed onto the pair constructor for the map.

Recipe 17.10: The push_range() function for queue, stack, and priority_queue

Problem: We want an easy way of adding several values at once to our container.

Solution: C++ 23 added the push_range() function.

This is another case where there was a workable solution before, but they added a feature to make things easier, and the code look better.

```
{

someNumbers{ 37 };

moreNumbers{ 65 };

st{ someNumbers };

<< "Stack starting with " << st.size();

st.push_range(moreNumbers);

<< ", and now has " << st.size() <<
```

```
//

q{ someNumbers };

<< "Queue starting with " << q.size();

q.push_range(moreNumbers);

<< ", and now has " << q.size() <<

}
```

In our code example, we initialize a std::stack with some numbers from a deque. Instead of looping through to add some more, we can add a range of values using the push_range() function. The following output shows how easy this works with std::queue as well as

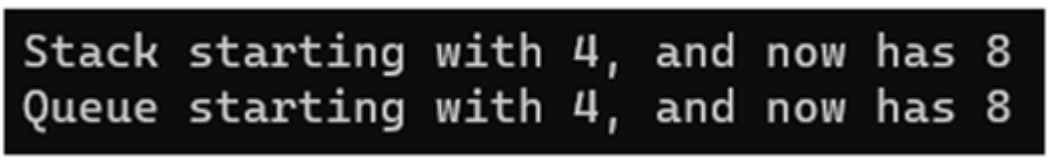

Figure 17.7: Output from LookAtPushRange

Recipe 17.11: What is std::mdspan

Problem: We want a clean way of working with multidimensional arrays

Solution: The C++23 release has

The std::mdspan (short for multidimensional span) is an exciting addition to C++ containers. The word span in the name is a key indicator. The mdspan does not own the contents, they must be supplied. There might be some cases that would seem like a negative, in terms of pre-allocating a container to provide the span. On the positive side, you can use the container for other purposes, such as other algorithms, and serialization.

When creating the you need to specify the dimensions. After that, the mdspan calculates the offsets required, and you can access the item by index/indices, with the Your mdspan can tell you the size of a dimension with the extent() function, by telling it the dimension index to get the size. As you can see in our example, you can even use the same source for multiple mdspan objects at the same time.

```
// .. add to other headers

#ifdef __cpp_lib_mdspan

#include
```

```
#endif

{

#ifdef __cpp_lib_mdspan

v =

auto ms2 =

<< "Two dimensional 2x4" <<

i != i++) {

j != j++) {


<< "[" << i << "," << j << "]=" << ms2[i, j] << "

    }

<<

  }

<<

<< "Three dimensional 2x2x2" <<
```

```
auto ms3 =

i != i++) {

j != j++) {

k != k++) {

<< "[" << i << "," << j << "," << k << "]=" << ms3[i, j, k] << "

          j, k] +=

      }

<<

   }

   }

<<

<< "Original vector, after modifications" <<

ii++) {

<< "v[" << i << "]=" << <<
```

```
    }

#endif

}
```

Our example creates two different The first is a two-dimensional and then a three-dimensional While printing our 3D we also modify the value. Then when we print out our original vector again, we see the updates are still there.

While std::mdspan may not be usable in all applications, it will be extremely handy to those doing computational science and other data handling.

If you look at the following output in Figure you can see our multidimensional spans printed out, along with our original vector showing the values were updated (using the mdspan indexes):

```
Two dimensional 2x4
[0,0]=4   [0,1]=42   [0,2]=99   [0,3]=37
[1,0]=17   [1,1]=27   [1,2]=33   [1,3]=65

Three dimensional 2x2x2
[0,0,0]=4   [0,0,1]=42
[0,1,0]=99   [0,1,1]=37
[1,0,0]=17   [1,0,1]=27
[1,1,0]=33   [1,1,1]=65

Original vector, after modifications
v[0]=1004
v[1]=1042
v[2]=1099
v[3]=1037
v[4]=1017
v[5]=1027
v[6]=1033
v[7]=1065
```

Figure 17.8: Output from LookAtMdspan

There are a few other interesting aspects about std::mdspan to mention: (See references)

If you do not know the dimension size at compile time, you can mark it as dynamic.

You can specify the layout property you want. You use specify layout_right to go left-to-right being row-focused, starting at the top-left item. You can also specify being column-focused, going top-down. There is also or you can define your own layout format.

You can also specify an accessor, meaning you can restrict in some cases, or compress memory, among other choices.

Conclusion

There seems to be a theme when looking at these container features: that it is not a groundbreaking feature that allows you to do something completely new. Instead, these features allow you to do what you could before, but with less code and more simplicity. These features mean less custom code, less loops, fewer calculations. Add in some performance gains as well, and it seems like good advancements for containers.

Points to remember

The and push_range() functions will be handy; so do not forget about them.

If you are doing a lot more lookups than additions to your map or set, you should switch to std::flat_map or

If you deal with multidimensional arrays, it will probably be of great benefit to check out

The C++23 push_range() method should be useful. It is always helpful when you get similar functionality, but do not have to have a for loop.

References

CppCon. (n.d.). Span.

YouTube. (n.d.). Mdspan.

Join our book’s Discord space

Join the book’s Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 18

Making it Easier to Code

Introduction

Apart from new features being added to make the language more powerful to use in new and different ways, many new features have been added to make what you do every day easier. We already saw the same about containers in the previous chapter. Most of the new classes and functions were not for doing something entirely new, rather, to make it easier for you to do the same things. This chapter will touch upon several features; whether it is formatting strings, working with numbers, or working with dates. The new standard library features help you get the job done.

Structure

In this chapter, we will cover the following topics:

Recipe 18.1: String formatting

Recipe 18.2: std::print

Recipe 18.3: starts_with and ends_with for strings

Recipe 18.4: Other enhancements for strings and ranges

Recipe 18.5: Using enum reduces typing for enums

Recipe 18.6: New date features

Recipe 18.7: New time features

Recipe 18.8: Timezone library

Recipe 18.9: std::midpoint

Recipe 18.10: includes pi and e

Recipe 18.11: Bit manipulation

Recipe 18.12: Designated initializer for aggregates

The exciting news is that a lot of these many new features are available for most compilers:

1.

2. {

3.

4.

5.

6.

7.

8.

9.

10.

11.

12.

13.

14.

15. y yes

16. }

Objectives

By the end of this chapter, the reader will find many ways to make their job a bit easier. You will learn there have been a lot of new smaller features added to recent releases that help you to code. You will learn about several new types for times and dates,, which will certainly reduce your effort if you deal with them a lot. You will learn of new mechanisms for enums, and initializers that are much appreciated. Plus, by the end of the chapter, you will know new ways for formatting, and outputting.

Recipe 18.1: String formatting

We want an easy way to format text.

C++20 has a new formatting library.

For many years, we have been yearning for something better when coding output statements. We liked some parts of but wanted it to improve its weaknesses, such as type safety. There are times when using std::cout is handy, but it could be very finicky and not clear how to create the output correctly. It certainly was not obvious with its modifiers; std::fixed (among many others) were sticky, meaning they are modifying all further parameters, or just the next parameter. With C++20's it solves many issues:

The simple fix of compile-time checking on the number of arguments, unlike having printf runtime exceptions.

std::format() has type safety with its arguments.

It is clear (at least for non-range values) about what modifiers are being used on a given parameter. So, there are no more questions on whether a modifier is sticky.

The format strings are more readable since you do not have values, format values, and format modifiers intermingled.

You can use positional arguments so that the parameters do not have to be in sequence.

You can create custom formatters.

With C++23, you can even use ranges as parameters for

Here is the list of header files we will be using in this chapter:

```
#include

#include

#ifdef __cpp_lib_print

#include

#endif

#include

#include

#include

#include
```

```
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include
```

#include

You can first review the code, and then we can talk about some of the details:

{

name;

};

template <>

: // to get a default parse function

{

&h, format_context& ctx) {

?

"I do not have a "ok to have more arguments than you :

"Hello, my name is {1}, I am h.age, h.name);

```
}

};

{



{

<< << << __FUNCTION__ <<

#ifdef __cpp_lib_format

<< answer to everything is

// compile time error, due to wrong number of parameters

// std::cout << std::format("The answer to everything is {}, but not {}.\n",
42);



<< C++{1} release is

<< this correct? {0}/{0:d} or
```

```
double num =

<< number = {0:.02}, or more precisely num);

<< something in between = num,

<<

<< {0:+}, {0:-}, {0: }, <<

<< {0:04X}, {0:#04x}, {0:b}, <<

<< {0:+}, {0:-}, {0: }, <<

<< {0:+}, {1:},

<<

<<

name = "C++

<< "pad left numbers" << ", result=" << <<

<< "pad right strings" << ", result=" << name) << "." <<

<< "pad to the right with character" << ", result=" << name) <<
```

```
<< "pad to the left with character" << ", result=" << name) <<

<< "centre with character" << ", result=" << name) <<

<<

#ifndef _MSC_VER

Helloer

Helloer 73 };

//std::string personFmt = «{}";

<< {} h1);

<< {0} h2);

#else

<< "Part of feature not yet implemented for this compiler." <<

#endif

#ifndef _MSC_VER
```

```
};

"The C++{1} release is

outputFile.close();

#else

<< "Part of feature not yet implemented for this compiler." <<

#endif

#if > 202207L

manyNumbers{ 67 };

// not working yet :-(


= "many numbers {}

<< manyNumbers);

#else

<< "Part of feature not yet implemented for this compiler." <<
```

```
#endif

#else

<< "Feature not yet implemented for this compiler at all." <<

#endif

}

};
```

We first use format with an empty format specifier. We then use numbered specifiers, and they do not have to be used in any particular sequence. The next line shows us the use of true and false, with the default output being text. However, we can also format it as a number. We then output a double value a few times, with a couple of different set precisions, and then we show how you can include the precision as an argument.

Our next code block looks at other number formats. We can prefix it with a sign or pad it with 0. We then print it in hex and binary. We also printed the Not-A-Number and infinity values. For the hexadecimal formatting, the example {0:04x} means:

The have learned, means using the first value in the argument list.

says to pad it with 0, if required, to a width of 4.

The says the value will be in hexadecimal. When we it meant the in uppercase. When we also specify it means that it will try to use an alternate output, and for a hex number, it means We also show binary values. There is also o for octal.

We then look at padding, showing how you can use < to fill the space on the right with a character, or > to fill the space on the left or use ^ to center it.

Lastly, we show how you can create a custom formatter for your type. You will need to define the format() and parse() functions (although since we were not parsing, we took a shortcut and derived from another type for the parse() function). Our formatter shows we have special output if the state is not correct. Otherwise, we can use the format_to() to send the output to the supplied context. Refer to Figure

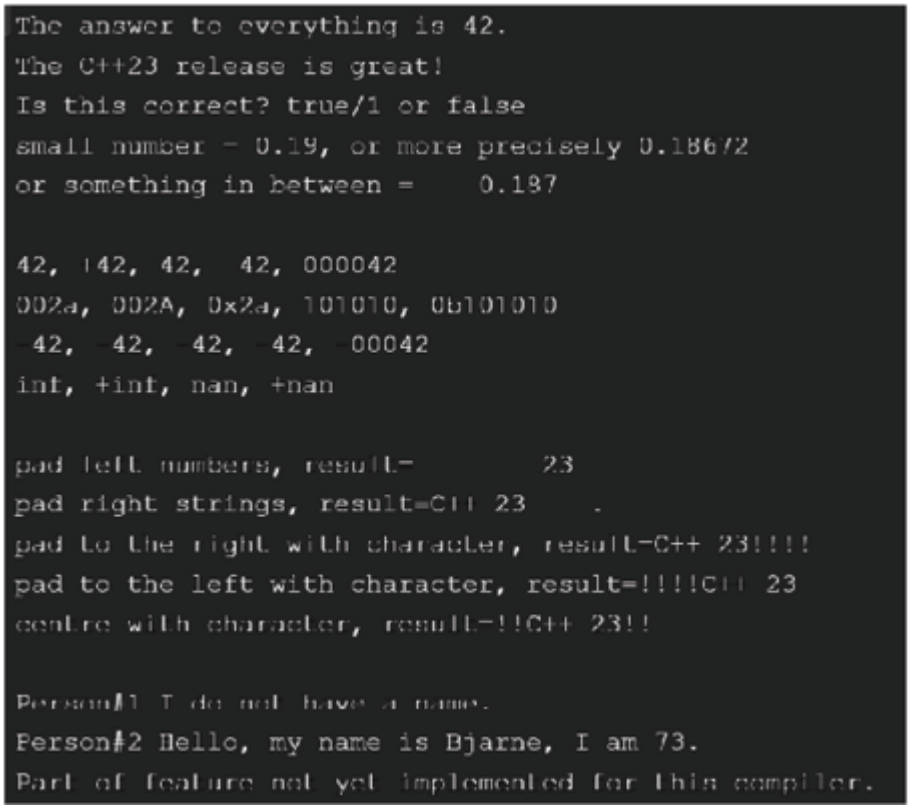
The answer to everything is 42.
The C++23 release is great!
Is this correct? true/1 or false
small number = 0.19, or more precisely 0.18672
or something in between = 0.187
42, 42, 42, 42, 000042
002a, 002A, 0x2a, 101010, 0b101010
-42, -42, -42, -42, -00042
int, +int, nan, +nan
pad left numbers, result= 23
pad right strings, result=C++ 23 .
pad to the right with character, result=C++ 23!!!!
pad to the left with character, result=!!!!C++ 23
centre with character, result=!!C++ 23!!
Person#1 I do not have a name.
Person#2 Hello, my name is Bjarne, I am 73.
Part of feature not yet implemented for this compiler.

Figure 18.1: Output for LookAtStringFormatting

There is a lot more functionality with std::format (see but a few more things to mention:

There are more specifiers for floating point numbers, such as e for scientific precision format (or E if you want an f of F for fixed precision formation, and g / G for general precision format.

The various compilers should allow for UTF8 strings.

There will probably be more custom formatters added in the future, but C++20 has made formatters for many of the std::chrono types, which we will learn about later in this chapter.

Recipe 18.2: std::print

We want a better way of outputting our text.

C++23 has a new method named std::print for text output.

We have just learned that C++20's std::format was a great addition. It modernizes the old printf() methodology and makes things a lot cleaner than with In the previous section, we had lots of examples where we created output text with std::format() and then sent it to You might ask how could it be any easier than that? The answer would be to combine those two things into a new std::print() function. The print() function will use the same formatting mechanism, except instead of outputting to a string, the output, by default, will be sent to the stdout stream. There is an overload where you can select your output stream, but this makes sending information to the standard output easier. There is also a related function named which does the same thing, except it sends a new line to the output as well. This is cleaner than having the code littered \n or std::endl entries.

Since it acts just like std::format() that we just talked about, there is not much new to show you, but here is a small code sample:

```
{
```

```
{
```

```
<< << << __FUNCTION__ <<

#ifdef __cpp_lib_print

C++{1} release is



C++{1} release is

};

"The C++{1} release is

outputFile.close();

#else

<< "Feature not yet implemented for this compiler at all." <<

#endif

}

};
```

There are a few more things to mention:

Since std::format() and std::print() were just created, they could utilize some new thinking and coding mechanisms. The result is that std::print will be significantly faster. In some cases, it could be perhaps 20%, but in other cases, it could be over 200% faster.

Another benefit of the new print() will be that it should have better Unicode support.

One can foresee that books and webpages that have a first lesson showing "Hello be a relic of the past:

Figure 18.2: Console output from LookAtPrint

Figure 18.3: File output from LookAtPrint

Recipe 18.3: starts_with and ends_with for strings

We want an easy way of seeing if our string starts or ends with some specific text.

C++20 has starts_with() and ends_with()

We saw in the previous chapter about containers, that the new contains() function works with some containers, as well as strings. There are times though, when you would want to know if your string starts or ends with some text. Thankfully in C++20, we can use that easily with starts_with() and The usage is obvious, so we can look at the code sample:

```
{



lhs, rhs) {

auto min = rhs.size());

#ifdef __cpp_lib_ranges_zip

// might be cleaner once we have zip/zip_transform

#endif
```

```
auto lPart = |

auto rPart = |

= lPart.begin();

= rPart.begin();

bool result =

for (; itL lPart.end() && itR != rPart.end(); ) {

if != {

    result =



  }

//

itL++;

itR++;
```

```
  }


}

{

<< << << __FUNCTION__ <<

message{ "C++23 is great" };

"C++" };

"great" };

"bad" };

<< "Message is: " << <<

<< (message.starts_with(startTerm) ? "Starts with: " : "Does not start with:
<< <<

<< (message.ends_with(endTerm) ? "Ends with: " : "Does not end with:
<< <<

<<
```

```
<< (message.starts_with(badTerm) ? "Starts with: " : "Does not start with:
<< <<

<< (message.ends_with(badTerm) ? "Ends with: " : "Does not end with:
<< <<



<<

"GREAT" };

<< (message.ends_with(upperEndTerm) ? "Ends with: " : "Does not end
with: << <<

<< (InSensitiveEndWith(message, upperEndTerm) ? "Case insensitive
Ends with: " : "Case insensitive does not end with: << <<

}

};
```

Our code does a few checks if it starts with and ends with a particular string. The thing to point out is they did not include an overload that requests a case-insensitive comparison. Even the newer string::contains() does not have that ability. For now, we created a helper function that leverages some range features. Using range functions, we reverse the string and take only the number of letters we are going to compare. We then convert them to uppercase for a fair comparison, stopping the loop if we see a mismatch. Refer to Figure

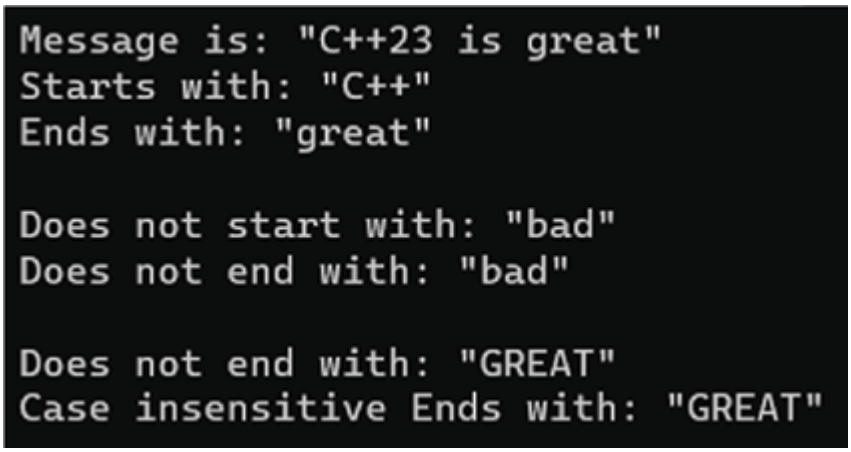
Message is: "C++23 is great"
Starts with: "C++"
Ends with: "great"
Does not start with: "bad"
Does not end with: "bad"
Does not end with: "GREAT"
Case insensitive Ends with: "GREAT"

Figure 18.4: Output from LookAtStartsWith

Our output shows that the start and end matching work fine, as long as you want it to be case-sensitive. If you need to be case-insensitive, then you will need to create some custom code.

Recipe 18.4: Other enhancements for strings and ranges

We want more ways to make using strings and ranges easier.

C++23 has a number of new member functions.

Since we use std::string so much in our everyday work, it is nice that they keep on finding ways to improve how we use them. Here are few more things that you can do when working with these classes, and once we start using ranges more, they will be helpful there too:

For containers such as vector and deque, we can assign (via move) a range of values.

Adds a range of values to the end. The container std::deque also has a prepend_range() function available.

For containers such as vector and deque, we can insert a range of values at a particular spot. The return value is an iterator to the insertion point, and the values provided must be movable.

This function is only for strings. You specify the start and end iterators, and a range that you will replace those values with.

A quick example of one of these would be:

```
{



{

<< << << __FUNCTION__ <<


goodMeals = "dinner" };

auto badMeals = "ice-cream" };

badMeals.assign_range(goodMeals);

<< "The bad meals are no longer bad:" <<

for {

<< meal <<

  }

}

};
```

And naturally, the output is as we would expect in Figure

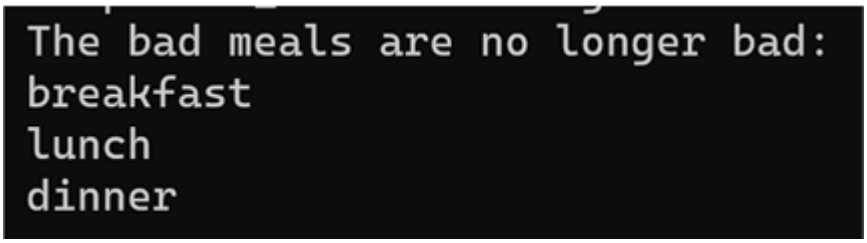

Figure 18.5: Output from LookAtRangeEnhancements

Recipe 18.5: using enum reduces typing for enums

We do not want to always type out the enum name when working with it.

C++20 has a new way that you can use enums within a block of code, with less typing.

C++ has had enums for a long time. They are simple and handy to use, but there are some things you should be careful of. If you look at the following sample code, we have defined a couple of enums. You see in the commented-out code that we tried to assign the wrong type to our enum instance, but thankfully the compiler saw that and gave an error. We then do a comparison between different types, and the compiler does not check that, as you can sadly see in the following output. We highly recommend you use scoped enumerations, just to avoid errors like this. It is a bit of work for existing code, for you to always scope the enum type for the enum values you are using, but it is much safer, and should be the default way of coding anything new you are working on.

{

{

<< << << __FUNCTION__ <<

```
breakfast, lunch, dinner };

fryPan, pot, wok, roastingPan };

= breakfast;

// most compilers will check is assigning to same type



// sm = wok;

if == SimpleCookware::fryPan) {

<< "eating a frying pan" <<

  }

breakfast, lunch, dinner, };

fryPan, pot, wok, roastingPan, };

// does not compile, needs better scope

// BetterMeal bm = dinner;

BetterMeal bm = BetterMeal::dinner;

// does not compile, since needs to specify enum type
```

```
// if (bm == wok) std::cout << "eating a work for dinner" << std::endl;

//

  {


// since enum is scoped in this block scope, can specify without being fully specific

  bm = lunch;

  }

// since the using statement is out of scope, needs to be fully specified

// bm = breakfast;

  {

switch (bm) {


case breakfast: << "eating breakfast:

case lunch: << "eating lunch:
```

```
case dinner: << "eating dinner:

  }

  }

eggs, pancakes, waffles, };

eggs, sandwich, soup, };


  {


// compiler will not allow this enum because it has a conflicting value
(eggs in both enums)

// using

  bd = eggs;

  }

  {
```

```
BreakfastDish::pancakes;

= BreakfastDish::waffles;

if == eggs) {

<< "3 eggs,

  }

== pancakes) {

<< "2 pancakes, with

  }

== BreakfastDish::waffles) {

<< "2 waffles, with

  }


  }

}

};
```

Refer to Figure

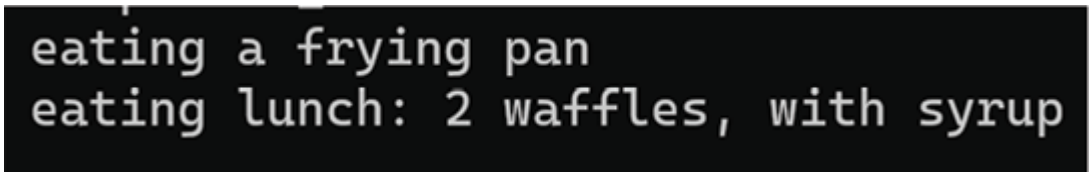

Figure 18.6: Output from LookAtEnums

Perhaps too many people said that, albeit safer, scoping the enum values was too much typing, so in C++20 they came up with a compromise. If you specify a using enum … in a block of code, you can specify an enum value without typing out the whole enum name. The compiler understands within that scope what enum values are acceptable and will allow you to do less typing. In the preceding sample code, you can see we had a using enum BetterMeal in a code block where we set the value, and in a switch block when comparing the value. What did not work, was when we tried to do multiple using enum statements that had overlapping names. You can definitely have multiple using enum statements. However, in our case, we have eggs as a value in both enums, and if we just specified eggs, then it may not know which enum type to use.

In our last code block, we did something differently. If, for some reason, you do not want to make all enum values available in short form for your code block, you can be explicit on which ones are. You can see in our if-else statements that, because of our using statement, we can specify the short form for eggs and pancakes, but we needed to be explicit and use the long form for waffles. This might be done if you have similar names in

different enum sets, and although (if you specify enum your compiler will know, the code might look better if you be explicit for some values.

One last note; you can see that we defined our enum with enum and then later defined another enum with enum They are equivalent so you can use whichever you choose.

Recipe 18.6: New date features

We want it to be easier to work with dates.

There is a lot of new functionality in C++20.

Since C++ is based on C, and C has been around since (almost) the epoch of time, there is a fair amount of baggage when it comes to dates and times. In terms of types, we have:

The type from the C library from the header named It is the number of seconds (but not counting the leap seconds) since midnight on January 1, 1970 UTC.

The similar named type std::time_t from the header in the C++ library. Not sure whether it is a wrapper, or a re-implementation, but it also does not count leap years, or leap seconds.

You have the struct tm, from the header in the C library. It has integers for each time period: along with a few others. The values are 0 based, except for tm_mday which is 1-31, tm_mon is 1-12, and the tm_year is the years since 1900.

Next is the similarly named std:tm from the C++ header It has the same members. Since it is just a struct, you need to 0 initialize it, or make sure you set all the values. You need to be extremely careful if trying to adjust it, because since it is a simple struct of integers, it has no logic about adjusting

other values. (If you adjust your hour by tm_hour -= then it does not adjust the day.

If you are working on a Microsoft Windows platform, then you may have also had to use types such as and

Then in C++11, we got the header that has some new types, new helper functions, and some typedefs.

In C++20, they added a lot to the std::chrono library, most new classes relating to clocks, calendars, and more. All of what was added recently could have been a chapter by itself. A lot of it is similar, and some of it is for specific scenarios, so we can just mention some parts.

This chapter is about making things easier to work on, and to accomplish that, we will try to ignore everything from the pre-chrono era.

Some of the recent date-related additions to the chrono library are:

The day class can be set and modified, and used for comparison (== and <=>). It also has an ok() function that checks if the value is between 1-31.

Can be set by number (January is 1), or by name. The month has several comparative operators, and an ok() function that checks if in the 1-12 range.

After setting or modifying it, you can use the ok() function to check if it is in the range of a short integer (±32767). The year class also has a simple is_leap() function.

This class can be set, modified, compared, and checked with an ok() function that checks for a value 0-6. It can use numbers, or predefined names. Sunday is 0, but if you can also set a value of 7. This means for those that have Monday as the first day of their week, you can iterate from Monday, which is 1 to 7 (but it actually stores Sunday as 0). The nice part about this is incrementing or decrementing the value loops around.

This class combines a weekday and an whose range of 1-5 can be checked if valid with the ok() function. This was created so you can specify things like the second Friday.

This class has a weekday member, and used when you need to specify that it is the last Thursday. Its ok() function just checks the validity of the weekday value.

This class has two members: obviously month and The ok() function checks whether both members are valid. Moreover, since it does not know the year, it will assume any February can have 29.

The ok() function can only check its month member value.

This class contains a and a weekday_indexed member, which are both checked when ok() is called. This is used when you need to specify something is on the Saturday.

This contains a and a weekday_last member, and the ok() function that looks at both of them. This is used for cases where you need to specify the last Monday of a month.

The ok() function checks both the year and month members.

This combines a and year member variable, which are all checked with the ok() function. It can now also check if a February is allowed for that specified year. It also has a couple of operators named sys_days and which are clocks that we will talk about shortly.

This class is constructed with a and a allowing it to now have a day, along with the month and year. The ok() can validate that the last day of February for 2024 is valid.

Similarly, this class is constructed with a and a This allows it to translate the Wednesday of June to now have a day member value and with the month and year.

As you would guess, this is built with a a and a This allows a full date to be composed.

If you look at the sample code for dates:

```
{



{



<< << << __FUNCTION__ <<



  month // 1-based
```

```
  month 10 };

assert(m == m2);

auto result = m +

assert(result == January);

  weekday // (sort-of) 0-based, starting on Sunday

  weekday

// assert(wd == m); // knows it cannot compare different types

assert(wd == wd2);

  wd += days{ 3 };

  wd2 = weekday{ 7 }; // also set to Sunday

assert(wd == wd2);

};

wdi =

  year yrsh{ 2112 };
```

```
yrsh,December, };

June, wdi };

mwd = { June, };

February }; // last day of the month

mdlast };

mdlast };



<< "Day for " << ymdLastLeap << " is " << ymdLastLeap.day() << ", is
leap? " << << ymdLastLeap.year().is_leap()

  << ",\n but day for " << ymdLastNoLeap << " is " << ymdLastNoLeap.day()
<< ", is leap? " << << ymdLastNoLeap.year().is_leap() <<

year_month_day June / / };

  year_month_day / June / }; // cannot do 6m, since could be confused with
minutes

year_month_day / / };

year_month_day / June / };

  year_month_day_last ymdLast{ / (January + / last };
```

```
September / 25 };

  md = { September / };

month_weekday April / };

October, weekday_last{Tuesday} };

mwdL = { December, Tuesday[last] };

/ mwdL };

=

{ February / / },

{ February / / },

{ December / / },

{ / mwdL},

{ ymwdL },

  };
```

```
auto Add4Days = year_month_day& ymd) -> sys_days { return
sys_days(ymd) + };

=

Add4Days),

=

for {

ymdOverAgain.push_back(d);

  }

for = i < ymdOver.size(); i++) {

<< ymdOver[i] << " +4= " << datesOver[i] << ", same as " <<

%B %d, ymdOverAgain[i]) <<

  }

auto now = system_clock::now();

};

<< "Today: day=" << ymdNow.day() << ", month=" << ymdNow.month() <<
", year=" << ymdNow.year() <<
```

```
<<

}

};
```

Note that so far we have tried to be clear and not use statements, but we are using a lot of types and the code would be littered with Thus, we are relaxing the book guidelines for this example.

First, we set a variable of type month to October, and one to the value of 10, and validate they are the same. Then we increase the variable by 3, and confirm it wrapped around to January.

The next block has us doing the same thing with the weekday type. We also try comparing a month and a weekday, and thankfully now the compiler knows that is not possible. When we add to our weekday variable, it wraps around as well.

We then create a weekday_indexed variable and set it in two ways: first, by using our previous weekday variable, then by using special syntax to state that we want the third Sunday.

After that, in the next code block we set the year, and use that year to initialize a We also define a month_weekday and set it using our previous as well an indexed weekday.

We then create a month_day_last values, and use that with a couple of year_month_day_last variables. Showing it now will tell us whether it is a leap year. If you look at the following output, you will see something interesting. When we print out the it actually shows the word last in the output, so that we know what it is.

The next block shows that we now have an easy and clean way to set dates. We can have no more ambiguity whether a value is a month or a day. For everybody to know what is a year, month, or day, we can use either the type or use a literal suffix of It is similar for the day as well. We need to use month{6} but not literal because that could be confused with minutes. It can make some assumptions, but it seems easy to be explicit. The other interesting part about setting these variables, is that we are using a slash as a separate for the date parts. We then set dates using and

Next, we create a few days with different constructors. We use those values using a range transform function to add some days and store that in a new vector of The entity sys_days is an alias for and we will talk more about time in the next section. For now, we will say that things such as year_month_day can store a valid value, but there is no context about time zone or where it is. We also have various clocks (such as that make things seem more real. When modifying there are times when you will have to use entities such as clocks. We do another loop, to create values in another just to show that we can easily convert a clock value back to a We print out that loop to show that clocks know about leap months, and how to roll over from December into the next year. We do not have to do any manual tweaks and validations. The other thing to note in the following output is how the date was printed. We use the std::format() command to get the day of the week and print the date as we want it.

The last code block gets the current date and time from the system clock, and then sets it to a year_month_day using the floor<> value. As you would guess, floor rounds down, and since we specified to round down to days, we just get the date part without the time. We then use the and year() functions to get the individual values. Refer to Figure

```
Day for 2024/Feb/last is 29, is leap? true,
 but day for 2023/Feb/last is 28, is leap? false
2024-02-28 +4= 2024-03-03, same as Sunday, March 03, 2024
2023-02-28 +4= 2023-03-04, same as Saturday, March 04, 2023
2024-12-28 +4= 2025-01-01, same as Wednesday, January 01, 2025
2024-12-31 +4= 2025-01-04, same as Saturday, January 04, 2025
2023-12-26 +4= 2023-12-30, same as Saturday, December 30, 2023
Today: day=18, month=Apr, year=2023
```

Figure 18.7: Output from LookAtDates

Recipe 18.7: New time features

We want it to be easier to work with times.

Slightly different question, but the same answer as previously: There is a lot of new functionality in C++20.

There are not as many new times relating to time, as there is relating to dates. For time-related things, there are 3 main classes:

This is not a class, but some requirements (possibly implemented with a concept) to denote what a clock must have. We will talk about those in a second (pun not intended).

This C++11 class specifies a point in time. You normally create it from a clock, or from a duration.

This C++11 class defines a time interval. It is often declared as the difference between two time_point values.

To be classified as a clock, meaning it would pass the chrono is_clock() it needs to satisfy the requirements:

You need to specify the arithmetic type/class used by the clock. This will help define its precision.

If you have used clocks or timers in coding before, then you have probably come across the idea of a tick. A tick is a small defined period of time. The period uses std::ratio to specify how long a tick is, in terms of seconds, or more likely what fraction of a second.

This uses rep and period to define the duration type.

Specifying the time_point type of the clock, relating to the epoch of when the clock started counting.

Relates to whether the clock advances at a steady rate, relative to real time.

To be a clock, you must have a function to get the current date and time.

There were already a few clocks, and a handful were added in C++20: (See references)

A C++11 class, relating to your system-wide realtime clock. This is probably the one you will use the most for normal date or time functionality.

This is a monotonic clock from C++11, meaning it will not jump backward or forward due to time adjustments or daylight savings time updates.

A C++11 class with the shortest tick period available. This is often used for timing events or how long something took.

This C++20 class relates to Coordinated Universal Time. This is currently the only clock (in the standard release) that tracks leap seconds. These are small adjustments (±1 second), normally once or twice a year, to keep the UTC clock set closer to what it should be.

This new class relates to International Atomic Time. This uses a weighted average of many atomic clocks to track time.

This C++20 class tracks the time maintained by the GPS satellites' atomic clocks.

This C++20 class is used by the file system for the timestamps on files. It does meet the clock requirements, but the epoch is not defined as it could vary between systems.

This C++20 class is called a pseudo-clock, since it relates to time on your system, but does not have a time zone specified.

You will mostly be dealing with one or two types of clocks, that behave similarly. You will just have to be aware of the precision and be careful when switching between different clocks. There is a clock_cast() function that allows you to convert a time point from one clock to another.

In terms of types, there are also two types of exceptions to be aware of, both of these related to daylight savings times. If you have a local_time that is modified during a daylight-saving time transition (which the time is being adjusted forward or backward one hour), then you can get an ambiguous_local_time exception. It means it was not sure whether the

result should factor in the daylight savings adjustment or not. Similar if you are converting a time during that transition period. There is also This relates to the transition when you lose an hour, and if you set your local_time for that period that does not exist, you will get an exception.

For our code sample, we again have a using statement to cut down on the number of times you see

```
{



{

<< << << __FUNCTION__ <<


auto now = system_clock::now();

= current_zone()->to_local(now);

<< "Now is " << now << " in UTC, or " << nowHere) << " for local" <<

auto durLocal = current_zone(), system_clock::now() }.get_local_time();

=

  hh_mm_ss hmsNow{ - daysTodayFloor) };
```

```
<< "Now is also " << hmsNow.hours() << ": " << hmsNow.minutes() <<
": " << hmsNow.seconds() <<

<<

auto later = nowHere +

<< "An hour later will be " << or later) <<

<<

=

= high_resolution_clock::now();


= high_resolution_clock::now();

  duration delta = timerEnd - timerStart;


<< "Took " << delta.count() << " ticks to sleep 1000ms" <<

= delta /

<< "Or took " << duration_cast(delta).count() << " milliseconds to sleep
1000ms"
```

```
  << ", which should be same as " << deltaMs <<

delta };

<< "And should be the same as: " << hms << <<

<<

= { };

for : listHours) {

= is_pm(itH);

<< "hour= " << itH << ", count=" << itH.count() << ", is AM? " << <<
is_am(itH) << ", is PM? " << itPm <<

  hours h12 =

  hours h24 = make24(h12, itPm);

<< "after make12: " << h12 << ", after make24: " << h24 <<

= itH +

<< "12 hours later: " << halfDayLater << << ", is AM? " <<
is_am(halfDayLater) << ", is PM? " << is_pm(halfDayLater) <<
```

```
<<

  }

}



};
```

In the first code block, we create a system_clock::time_point via the now() function. We then convert it to a local_time variable using our current time-zone. We will be talking about time zones in the next section. We then print out our times, either with default formatting, or by using

The next code block shows us creating a local_time variable by first creating a zoned_time value, and then calling get_local_time using our current time-zone. We then get the start of that day by using the floor<> function again. The difference between the original time and the floor time initializes our hh_mm_ss variable. We print that out with individual calls to and

We then show that we can take the local_time by easily and clearly adding an hour to it.

The high_resolution_clock is then used to time an operation. The result of which we create a duration value and print that out in a few ways, including creating a hh_mm_ss value.

The last code block iterates over a list of hours values. We print out whether it thinks it is AM or PM, and also convert it to 12- or 24-hour time. If you look in the following output, the first couple of values work fine, but then when we are moving from 12 hours to 13 hours, it does not know if it is AM or PM anymore. That is just to show that since you can use hours (or any unit) to measure something, then it will not wrap around. Refer to Figure

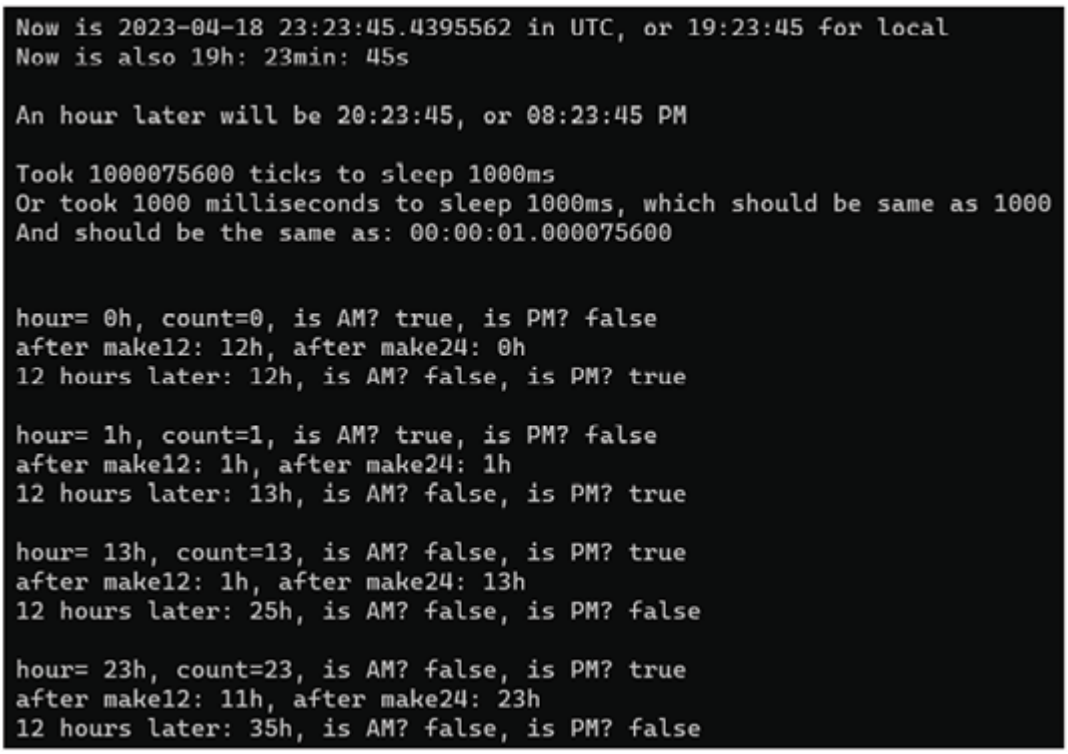
Now is 2023-04-18 23:23:45.4395562 in UTC, or 19:23:45 for local
Now is also 19h: 23min: 45s
An hour later will be 20:23:45, or 08:23:45 PM
Took 1000075600 ticks to sleep 1000ms
Or took 1000 milliseconds to sleep 1000ms, which should be same as 1000
And should be the same as: 00:00:01.000075600
hour= 0h, count=0, is AM? true, is PM? false
after make12: 12h, after make24: 0h
12 hours later: 12h, is AM? false, is PM? true
hour= 1h, count=1, is AM? true, is PM? false
after make12: 1h, after make24: 1h
12 hours later: 13h, is AM? false, is PM? true
hour= 13h, count=13, is AM? false, is PM? true
after make12: 1h, after make24: 13h
12 hours later: 25h, is AM? false, is PM? false
hour= 23h, count=23, is AM? false, is PM? true
after make12: 11h, after make24: 23h
12 hours later: 35h, is AM? false, is PM? false

Figure 18.8: Output from LookAtTimes

Recipe 18.8: Timezone library

We want an easy way to work with time-zones.

C++ 20 created new functionality for that.

You might not use time-zones too often, but the world is getting smaller, and we are dealing with people, servers, and goods from different parts of the globe. C++ has made it easier by providing new clocks as we mentioned before, and new classes that relate to time-zones. We have already mentioned a few already, but we will go through an example and describe it more. Obviously, it would be handy to have a list of time-zones, but the good and bad of it is that there is not a single list to use. Unix/Linux systems have traditionally used something named the IANA database, whereas newer Windows systems use the ICU library. The ICU library was based on IANA, and while it is not as thorough, it is being updated via the Windows Update utility. C++ covered the bases but accessing the get_tzdb_list() function returns all available databases, and you can use the one that is available to you. From there on, the usage is the same. You can retrieve your own with a call to and iterate through the others, or look up one in particular.

The following code example shows us checking what other time-zones have the same UTC offset in minutes as we have:

```
{



{
```

```
<< << << __FUNCTION__ <<

auto now =

=



auto myMins = /

othersInSameTz{};

for itDb = itDb != itDb++) {

<< "zone=" << itDb->version <<

for it = it != itDb->zones.end(); it++) {

if == it->name()) {

<< "Here I am: " << it->name() << "  !!!!!!!!! "

      << myMins << " from UTC" <<

    }

else {

auto thereMins = /
```

```
if == thereMins) {

othersInSameTz.push_back(it->name());

    }

    }

  }

auto joinedCharacters = '',

<< "Others in same timezone:


for << c;

<< <<

<< "Local time:" << myZone->name(), now } << '', offset=" << myMins <<

try

  {

tzIST =

<< tzIST, now } << " for " << tzIST->name() <<
```

```
auto offsetIST = /

= offsetIST - myMins;

tzDelta };

<< "Difference between local time and IST=" << tzDelta << ", " <<
tzDeltaHms <<

  }

catch ex)

  {

<< "Problem checking timezone. " << ex.what() <<

  }

  }

}

};
```

Looping through the different time-zones, we use the same time_point to find out the UTC offset. We then use a range function to concatenate the names

and print them out, as shown in Figure

zone=2021a.27
Here I am: America/Toronto !!!!!!!!!! -240min from UTC
Others in same timezone: America/Anguilla, America/Antigua, America/Aruba, America/Asuncion, America/Barbados, America/Blanc-Sablon, America/Boa_Vista, America/Campo_Grande, America/Caracas, America/Cuiaba, America/Curacao, America/Detroit, America/Dominica, America/Grand_Turk, America/Grenada, America/Guadeloupe, America/Guyana, America/Havana, America/Indiana/Marengo, America/Indiana/Petersburg, America/Indiana/Vevay, America/Indiana/Vincennes, America/Indiana/Winamac, America/Indianapolis, America/Iqaluit, America/Kentucky/Monticello, America/Kralendijk, America/La_Paz, America/Louisville, America/Lower_Princes, America/Manaus, America/Marigot, America/Martinique, America/Montreal, America/Montserrat, America/Nassau, America/New_York, America/Nipigon, America/Pangnirtung, America/Port-au-Prince, America/Port_of_Spain, America/Porto_Velho, America/Puerto_Rico, America/Santiago, America/Santo_Domingo, America/St_Barthelemy, America/St_Kitts, America/St_Lucia, America/St_Thomas, America/St_Vincent, America/Thunder_Bay, America/Tortola, EST5EDT, Etc/GMT+4, SystemV/AST4, SystemV/AST4ADT
Local time:2023-04-18 21:11:08.1160069 EDT, offset=-240min
2023-04-19 06:41:08.1160069 GMT+5:30 for Asia/Calcutta
Difference between local time and IST=570min, 09:30:00

Figure 18.9: Output from LookAtTimezone

There is other information in the time-zone database as well. It has a list of links, or short names for the time-zones, as well as the list of leap_seconds that it knows about. There are other new classes related to leap seconds, but since that is probably not your focus right now, we will skip that.

The last code block looks up a time-zone with We then get the UTC offset for that in minutes and compare that to our current time-zone. We also use zoned_time to convert our UTC to the given time zone. The difference in the current times matches up with the time-zone difference.

Recipe 18.9: std::midpoint

We want an easy way of getting the midpoint, or average between two values.

There is a new function in C++20 named

Upon first glance, you would not think this is a complex enough problem to warrant a function. Most often, you will see (a+b)/2 in code. The problem is that it seems too easy, and you do not think about any overflow issues. Luckily, other people have thought about that, to avoid overflow exceptions, or worse, to unknowingly get wrong results. This seemingly simple function will safely give the value you want:

```
{



{

<< << << __FUNCTION__ <<

bunchesOfInts{

 , - };

bunchesOfLongLongs{

 , - };
```

```
bunchesOfDoubles{ };

for {

<< "Mid point of " << pair.first << ", and " << pair.second << " = " <<
pair.second) <<

  }

<<

for {

<< "Mid point of " << pair.first << ", and " << pair.second << " = " <<
pair.second) <<

  }

<<

for {

<< "Mid point of " << pair.first << ", and " << pair.second << " = " <<
pair.second) <<

  }
```

```
}

};
```

This gives us the following results:

```
Mid point of -42, and 42 = 0
Mid point of 1, and 2 = 1
Mid point of 0, and 100 = 50
Mid point of -2147483648, and 2147483647 = -1
Mid point of 2147483647, and 2147483645 = 2147483646

Mid point of -42, and 42 = 0
Mid point of 0, and 100 = 50
Mid point of -9223372036854775808, and 9223372036854775807 = -1
Mid point of 9223372036854775807, and 9223372036854775805 = 9223372036854775806

Mid point of -42, and 42 = 0
Mid point of 222.222, and 444.444 = 333.333
Mid point of 2.22507e-308, and 1.79769e+308 = 8.98847e+307
```

Figure 18.10: Output from LookAtMidpoint

Most of these seem straightforward. Since the result is the same type as the inputs, we do not get fractions and it rounds integers down. The nice part is when we look for the midpoint of a couple of near-maximum values and we see the result without any problems.

One other point of interest, the std::midpoint() function is called an interpolation operation. There was another interpolation operation added in C++20 named This computes the linear interpolation value (or linear extrapolation value in some cases). It is overloaded to use values of type float, double, or long double.

Recipe 18.10: includes pi and e

We want an easy way of using some mathematical constants.

C++ 20 created a new header with many values.

We continue our theme about numbers by talking about the new header and the values in the std::numbers namespace. This means you can get better precision than what you might have with some of your own defined constants. You also do not have to use a third-party library just for a simple value either. You can use the predefined constexpr double value or use the template for your precision. These constants include the following information, as shown in Table

Table 18.1: New constants and templates

For the following code example, we tried a few simple equations:

```
template T>

T r, T pi) { * r; }

template T>

T r, T pi) { * r * pi; }

template T>
```

```
T r, T pi) { return r * r * pi; }

{


{

<< << << __FUNCTION__ <<


  <<

  << "float:       " << <<

  <<


  << "double:      " << <<

  <<

  << "long double: " <<

  << <<

#ifdef _MSVC_LANG
```

```
radii{ };

for : radii) {


   <<

   << "r=" << r

   << ", diameter=" << DiameterFn(r,

   << ", circumference=" << CircumferenceFn(r,

   << ", area=" << AreaFn(r, <<

  }

#endif

}

};
```

You can see in the following output in Figure the difference in precision between float and the doubles. It was nice to see for the last value in our loop that we got an area of 1. Having that extra precision improves the accuracy of the results.

```
float:       3.14159274
double:      3.1415926535897931
long double: 3.1415926535897931

r=1, diameter=2, circumference=6.28318530717959, area=3.14159265358979
r=2, diameter=4, circumference=12.5663706143592, area=12.5663706143592
r=3, diameter=6, circumference=18.8495559215388, area=28.2743338823081
r=0.564189583547756, diameter=1.12837916709551, circumference=3.54490770181103, area=1
```

Figure 18.11: Output from LookAtNumbers

Recipe 18.11: Bit manipulation

We want an easy way to work with bits.

The C++20 and C++23 releases have many new functions to use.

Another new header is which contains many new handy functions when dealing with bits and bit logic. If you want to simply swap byte values of an integer type, you can use C++23's

```
template T>

n, {

<< << <<

  << * << n << " (

for i != ++i, n >>= {

<< << & n) <<

}

<< ")" << << (endLine ? "\n" :
```

```
}

template T>

title, T n) {

<< title << " :

DumpBytes(n,

<< "byteswap:



<<

}

{



{

<< << << __FUNCTION__ <<
```

<<

}

//...

};

Refer to Figure

```
16 bits : 0123 ( 23 01 )
byteswap: 2301 ( 01 23 )
32 bits : 01234567 ( 67 45 23 01 )
byteswap: 67452301 ( 01 23 45 67 )
64 bits : 0123456789ABCDEF ( EF CD AB 89 67 45 23 01 )
byteswap: EFCDAB8967452301 ( 01 23 45 67 89 AB CD EF )
```

Figure 18.12: Output from LookAtBits_byteswap

A fast way to check if your integer is a power of two, you can use std::has_single_bit() from the C++20 release that checks if only one bit is true:

{

// ...

{

```
<< << << __FUNCTION__ <<

numbers{ };

for : numbers) {

<< n << ": has_single_bit( " << << " )= "

   << << <<

 }

<<


};
```

Refer to Figure

```
0: has_single_bit( 00000000 )= false
1: has_single_bit( 00000001 )= true
2: has_single_bit( 00000010 )= true
3: has_single_bit( 00000011 )= false
4: has_single_bit( 00000100 )= true
7: has_single_bit( 00000111 )= false
8: has_single_bit( 00001000 )= true
16: has_single_bit( 00010000 )= true
31: has_single_bit( 00011111 )= false
32: has_single_bit( 00100000 )= true
```

Figure 18.13: Output from LookAtBits_has_single_bit

To find out the smallest integral power two that is not smaller than a value, you can use It is a bit like rounding up to the next power of 2. Similarly, std::bit_floor() returns the largest integral power of 2 that is not greater than a value. You can also think of it as rounding down to the nearest power of 2, or basically saying what is the highest bit that was set for this number.

```
{



// ...

{

<< << << __FUNCTION__ <<

numbers{ };

for : numbers) {

auto n2c =

auto n2f =

<< "for " << n << ": bit_ceil( " << << " ) = "

    << << n2c << " or " <<
```

```
    << ", bit_floor = «

    << << n2f << " or " <<

    <<

  }

<<

}

//...

};
```

Refer to Figure

```
for 0: bit_ceil( 00000000 ) = 1 or 00000001, bit_floor = 0 or 00000000
for 1: bit_ceil( 00000001 ) = 1 or 00000001, bit_floor = 1 or 00000001
for 2: bit_ceil( 00000010 ) = 2 or 00000010, bit_floor = 2 or 00000010
for 3: bit_ceil( 00000011 ) = 4 or 00000100, bit_floor = 2 or 00000010
for 4: bit_ceil( 00000100 ) = 4 or 00000100, bit_floor = 4 or 00000100
for 7: bit_ceil( 00000111 ) = 8 or 00001000, bit_floor = 4 or 00000100
for 8: bit_ceil( 00001000 ) = 8 or 00001000, bit_floor = 8 or 00001000
for 16: bit_ceil( 00010000 ) = 16 or 00010000, bit_floor = 16 or 00010000
for 31: bit_ceil( 00011111 ) = 32 or 00100000, bit_floor = 16 or 00010000
for 32: bit_ceil( 00100000 ) = 32 or 00100000, bit_floor = 32 or 00100000
```

Figure 18.14: Output from LookAtBits_bit_ceil_bit_floor

We also have helpful that tells us how many bits we need to store our value:

```
{


// ...

{

<< << << __FUNCTION__ <<

numbers{ 255 };

for : numbers) {

<< "for " << n << ": bit_width( " << << " ) = "

   << <<

  }

<<

}

//...

};
```

Refer to Figure

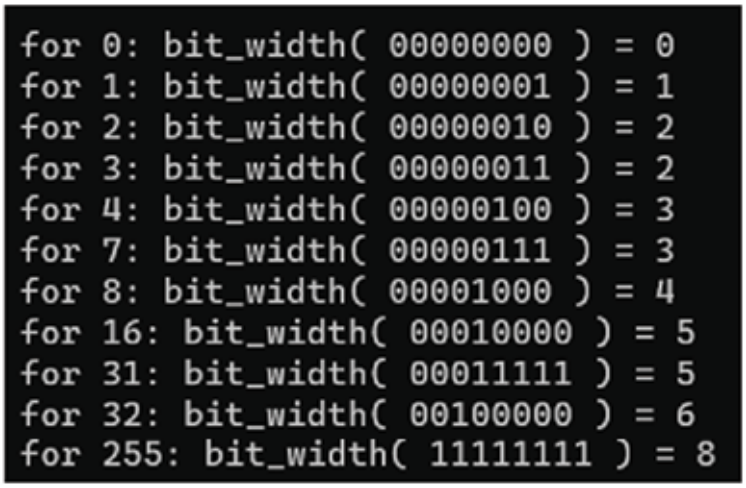
for 0: bit_width(00000000) = 0
for 1: bit_width(00000001) = 1
for 2: bit_width(00000010) = 2
for 3: bit_width(00000011) = 2
for 4: bit_width(00000100) = 3
for 7: bit_width(00000111) = 3
for 8: bit_width(00001000) = 4
for 16: bit_width(00010000) = 5
for 31: bit_width(00011111) = 5
for 32: bit_width(00100000) = 6
for 255: bit_width(11111111) = 8

Figure 18.15: Output from LookAtBits_bit_width

Our next code block shows std::rotl() and which shift the value by the given number of bits. We printed the value several times, just to make it easier to compare the value with the one shifted below. Note that if you do a shift left a negative number of times, that is the same as shifting right a positive number of times. The other important part to note is these operations do a circular shift, so no bits are lots.

{

// ...

{

<< << << __FUNCTION__ <<

n =

```
<< "n      = " <<

  << ", n        = " <<

  << ", n        = " <<


  << ", n      = " << <<

numbers{ };

for : numbers) {

= shift);

= -shift);

= -shift);

= shift);

<< "rotl(n," << shift << ")=" <<

    << ", rotr(n," << -shift << ")=" <<

    << ", rotl(n," << -shift << ")=" <<

    << ", rotr(n," << shift << ")=" <<
```

```
    <<
  }
<<
}
// ...
};
```

Refer to Figure

```
n       = 01101001, n        = 01101001, n        = 01101001, n       = 01101001
rotl(n,1)=11010010, rotr(n,-1)=11010010, rotl(n,-1)=10110100, rotr(n,1)=10110100
rotl(n,2)=10100101, rotr(n,-2)=10100101, rotl(n,-2)=01011010, rotr(n,2)=01011010
rotl(n,4)=10010110, rotr(n,-4)=10010110, rotl(n,-4)=10010110, rotr(n,4)=10010110
rotl(n,9)=11010010, rotr(n,-9)=11010010, rotl(n,-9)=10110100, rotr(n,9)=10110100
```

Figure 18.16: Ooutput from LookAtBits_rotl_rotr

There are times when you need to count how many 0s are at the start of your number, or how many 1s are at the end of your number. You can use or countr_one() to quickly find that out.

```
{
// ...
```

```
{

<< << << __FUNCTION__ <<

numbers{ 0b11111111 };

for : numbers) {

= n;

<< iN << " or 0b" << << ": "

    << "countl_zero=" <<

    << ", countl_one=" <<

    << ", countr_zero=" <<

    << ", countr_one=" << <<

  }

<<

}

// ...

};
```

Refer to Figure

```
0 or 0b00000000: countl_zero=8, countl_one=0, countr_zero=8, countr_one=0
20 or 0b00010100: countl_zero=3, countl_one=0, countr_zero=2, countr_one=0
235 or 0b11101011: countl_zero=0, countl_one=3, countr_zero=0, countr_one=2
255 or 0b11111111: countl_zero=0, countl_one=8, countr_zero=0, countr_one=8
```

Figure 18.17: Output from LookAtBits_countl_countr

You can quickly get a count of the 1 bits in your number by using

```
{

// ...

{

<< << << __FUNCTION__ <<

numbers{ 0b11111111 };

for : numbers) {

<< "popcount( " << << " )= "

   << <<

 }
```

<<

}

// ...

};

Refer to Figure

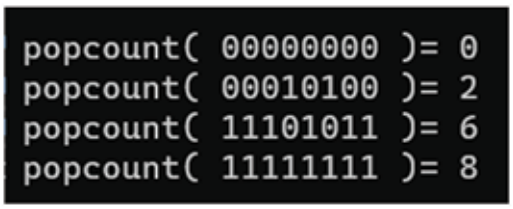

Figure 18.18: Output from LookAtBits_popcount

There are times when you need to know the endianness of your system. Windows systems tend to use Little-endian, while others might use mixed. There is now an enum that you can check:

{

// ...

{

```
<< << << __FUNCTION__ <<

if == {

<< "current system is using

  }

== {

<< "current system is

  }

else {

<< "current system is

  }

<< <<

}

// ...

};
```

Refer to Figure

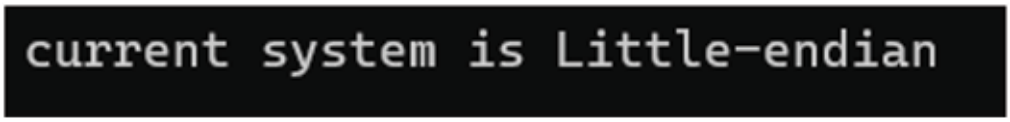

Figure 18.19: Output from LookAtBits_endian

The last bit of code shows allowing you to convert values by reinterpreting the bits. It is possible to even reinterpret complex structures, as long as the rules are followed for both types (to and from). You cannot use a union, or pointer type, or pointer to a member, or volatile, or static reference member. It is almost like a std::memcpy but with extra checks/restrictions.

```
{

// ...

{

<< << << __FUNCTION__ <<

double f64v =

auto u64v =

auto f64vAgain =


  << "std::bit_cast(" << << f64v << ") == 0x"
```

```
  << << u64v

  << ", back to f64=" << << f64vAgain <<

<<

}

// ...

};
```

Refer to Figure

```
std::bit_cast<std::uint64_t>(10501430590.000000) == 0x42038F78E9F00000,
back to f64=10501430590.000000
```

Figure 18.20: Output from LookAtBits_bit_cast

Recipe 18.12: Designated initializer for aggregates

We want an easy way of initializing an aggregate.

In C++20, they provided a new way of doing that.

It has been possible since C++11 to use a designated initializer. This meant you could initialize your variables, even containers:

```
numbers1 = { };

numbers2 { };
```

You can also initialize a struct or class's member variables (with some restrictions) as long as you place the values in the order of the variables. The problem with that is that it is not clear which variable you are initializing.

With C++20, there is a new way that in the initializer block, you can use a .MemberName syntax to make it clear which member variable you are initializing. You can see in the following code example, that you can still initialize with or without the equal sign. For some reason, the shortcoming still exists, meaning that the variables have to be initialized in sequence of declaration, but at least the code is more understandable now. Note that you can still initialize without declaring with variable, although personally, the pain of specifying the variable names is small compared to the increased readability.

{

numbers1;

numbers2;

};

{

numbers3;

message;

numbers4;

};

{

{

<< << << __FUNCTION__ <<

```
numbers1 = { };

numbers2{ };

  strictDI2 s3{

// member variables cannot be initialized out of order

// .message = «Hello C++",

.numbers3 { },

.message = "Hello

.numbers4 = { },

.innerDI {

.numbers1 {

.numbers2 {

  },

  };

<< "s3= " << s3.message <<
```

```
 strictDI2 s4{

 "Hello

 };

<< "s4= " << s4.message <<

}

};
```

For now, there are still a number of restrictions for initializing an aggregate. The worst is that you cannot have a protected base class, or inherited constructor. Hopefully, that can be remedied one day. It also cannot have a constructor, but that does not seem to be a hassle, since if it had one then you would be using it anyway.

Refer to Figure

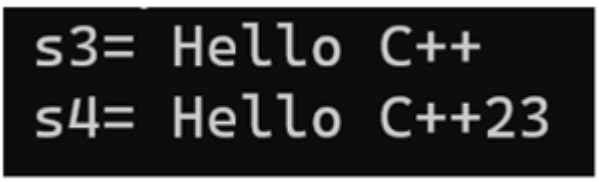

Figure 18.21: Output from LookAtDesignatedInitializer

Conclusion

At the beginning of the chapter, we said that you will learn some things that will make coding easier. We certainly mentioned a lot of new classes and functions, so hopefully some of that you will find to be of great help. Previous chapters have certainly talked about new features to make your coding life simpler. Adding constexpr to your code consists of more typing, but finding issues at compile-time versus at runtime certainly sounds like less stress when creating code. Using coroutines or some of the new containers means working on something different, but the result should mean that it is easier to accomplish what you want. You should use as much of the new ranges functionality as you can, since that definitely will mean less code you need to create, and it should look a lot nicer as well. Using the new format and print functions we just learned, should be a welcoming addition. Even if they are small topics such as starts_with or the contains functions, they should put a smile on your face. Whether it is the new functions relating to bits, or the new number templates, recent C++ releases have done a lot to help us out.

Points to remember

The format and print functions use similar formatting, and so it would be good to get an understanding of how you want to format the values because you can use that in a lot of places.

There are a lot of new time and date classes to learn how to use, but you should be rewarded when you find how much clear and straightforward your code will be.

With relating to clearer code, using the new aggregate initializer will have your code less error prone, as well as looking a lot better. You just need to remember that despite the member values being named, you still need to put them in the same order as their declaration.

References

YouTube. (n.d.) Clocks. Retrieved from https://youtu.be/Dq7rqjatxz8

Wikipedia. (n.d.) Endianness. Retrieved from https://en.wikipedia.org/wiki/Endianness

YouTube. (n.d.) Explicit. Retrieved from https://youtu.be/Rr1NX1lH3oE

CppReference. (n.d.) Formatting. Retrieved from https://en.cppreference.com/w/cpp/utility/format/formatter

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 19

Making Your Code Cleaner

Introduction

You might think this is a catch-all (or chapter, but these new language features will help you write better code. Sometimes it is the little things that make a big difference. There have been many features that we have talked about in previous chapters, which will make your code cleaner. The features here may not fit into another chapter, so yes, it is a bit of a catch-all. Hopefully, some of these features will allow you to make your code neater.

It is believed that the best situation is when you are writing code so others can learn from you. It is not about some other architecture or project documents in some folder. The best way others can learn is if you write good, clean code that others can follow. There are other benefits to this stage as well. You might get more recognition, and you might get more rewards.

This chapter is a collection of new C++ features. Perhaps you will be able to use some of these in your daily coding. Hopefully, you also see that these features should make your code cleaner and easier for you to understand, and for others to understand.

Structure

In this chapter, we will cover the following topics:

Recipe 19.1: Familiar Template Syntax Lambdas

Recipe 19.2: Lambda parameter packs

Recipe 19.3: Literal class types in non-type template parameters

Recipe 19.4: Multidimensional subscript operator

Recipe 19.5: std::make_shared supports arrays

Recipe 19.6: CTAD improvements

Recipe 19.7: Deducing this

Recipe 19.8: Transforming auto

Recipe 19.9: char8_t and std::u8string types

Recipe 19.10: Suffix for std::size_t

Recipe 19.11: Named universal character escapes

Recipe 19.12: Monadic operations for std::optional

Recipe 19.13: Using std::expected

Recipe 19.14: Pre-processing directives elifdef and elifndef

Recipe 19.15: invoke_r

Recipe 19.16:

Recipe 19.17: Bit initialization

Recipe 19.18: Extend init-statement to allow alias-declaration

Since this chapter is a collection of many different features, we get many different results whether the compiler has completed that feature. Overall, there is decent coverage, but a few that we will still have to wait for.

{

}

Objectives

By the end of this chapter, the reader will understand some new ways to write clean code, and why it is important. Some of the new features you will learn are structural, while other features are new types, or new keywords. Admittedly some are easier than others, but after you have learned them, then you will appreciate how they can help you.

Recipe 19.1: Familiar template syntax Lambdas

Problem: We want a cleaner way of using lambdas.

Solution: C++20 has created a way to make templated lambdas look better.

Lambdas have been a part of the language since C++11. Each release has had some additions. For C++20, they added a mechanism so that knowing the deduced type a lot easier. The result is a cleaner look with less code and looking more natural. The auto keyword is handy, but then you need to get the type with Aside from the bit of extra typing, it does not look clean, and certainly is not as handy.

First, let us state the headers we will be using in this chapter:

```
#include

#include

#include

#include

#include
```

```
#include

#include

#include

#include

#include

#include

#include

#include

#include

#include

#include

#include

#include // for bit manipulation

#include
```

```
#include

#include

#include

#include

#include

#ifdef __cpp_lib_spanstream

#include

#endif

#include

#include

#include

#include

#include
```

If we look at the code snippet:

```
{



{

<< << << __FUNCTION__ <<

auto Cpp14AutoLambda = x, y) {



};

auto Cpp20TemplatedLambda = [] X, class && x, Y && y) {



  };

}

};
```

It does not show a huge difference, but the second lambda does look better. (see references) We are used to the syntax of using < > to template our code, and so the option of specifying a type and not using auto is familiar and makes the lambda seem natural.

Recipe 19.2: Lambda parameter packs

Problem: We want a cleaner way of using a variadic template with lambdas.

Solution: C++20 added a cleaner syntax for parameter packs.

While we are talking about lambdas, we can mention another feature as well. Parameter packs were also released with C++11, and they allow you to provide an unspecified number of parameters to a templated function.

We can do a quick review with the following sample code. Our Lcm() function has a parameter pack. Although it could just say we specified that the parameters need to be some type of integers. There are many ways you can work with parameter packs; in this case, we say our method has at least two arguments. We get the least-common-multiplier for the first two values, and then check if we have any more in the rest parameter. If the rest parameter is empty, then we are done. If we do have more, then we can recurse and call our function again, with the result we just calculated as the first parameter. This will make the parameter expand, taking a value out of rest to be passed into the second parameter. We keep doing this until we have no more values to evaluate. This is a clean and efficient way of working through an unspecified number of parameters.

{

TIntegral, std::integral... TRest>

```
static constexpr TIntegral TIntegral& first, const TIntegral& second, const TRest& ... rest) {

result = std::lcm(first, second);



> {

rest...);

  }

return result;

}

Rest>

static auto LcmWithLambda(Rest... rest) {

  auto lcmFn = [&, rest...] {



  };

  auto res = lcmFn();
```

```
return res;

}


static void LookAtTemplateExpansion() {

std::cout << std::endl << std::endl << __FUNCTION__ << std::endl;

std::cout << "result1=" << << std::endl;

std::cout << "result2=" << << std::endl;

std::cout <<

}

static void LookAtLambdaExpansion() {

std::cout << std::endl << std::endl << __FUNCTION__ << std::endl;

LookAtTemplateExpansion();


std::cout << "result from calling lambda=" << << std::endl << std::endl;
```

```
}

};
```

C++20 added lambda captures for parameter packs. If we had a requirement that we needed a lambda, we can define the lambda with a variadic capture, and call the Lcm() function we used previously. We can see our LcmWithLambda function still has similar functionality, it is now just a bit cleaner.

This gives us the same result, as shown in Figure

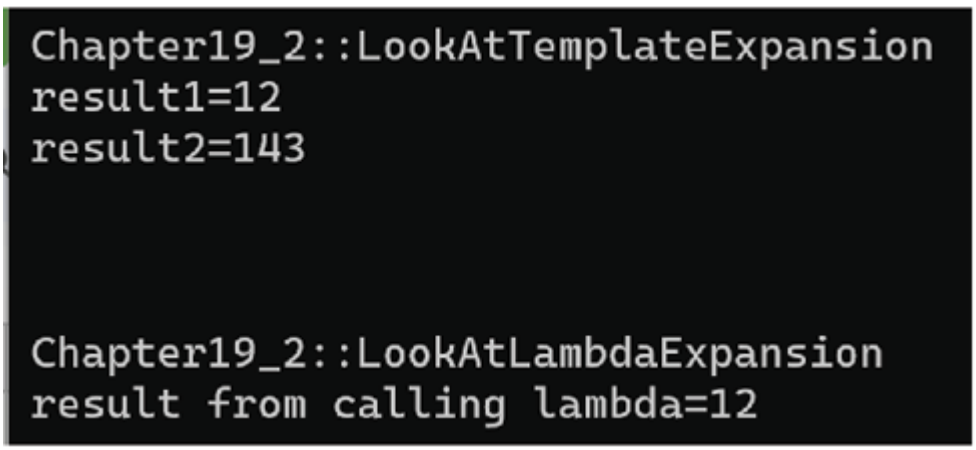

Figure 19.1: Output from LookAtLambdaExpansion

Recipe 19.3: Literal class types in non-type template parameters

Problem: We want a way of passing different types of data into our templates.

Solution: C++20 made more allowances for that.

The topic will probably need a couple of explanations. We have used a number of templates, so that part should be clear. We can pass non-type parameters into a template, meaning something that is not templated, but rather, is an actual value. These values need to have a structural type. This means they could be:

An enum

An integer

A float

A pointer (to an object or function)

A l-value reference (to an object or function)

What C++20 allowed us to do is to pass in a class type, meaning an instance of a class. We did add the extra qualifier of literal class indicating that it cannot be any instantiation of a class. It would need the following properties:

All non-static members (in itself, and any base class) would need to be public and non-mutable. This could mean that everything is constant.

All non-static members (in itself, and any base class) would need to be structural types.

So, although there are quite a few restrictions, this adds some new capabilities. If you were looking for such a mechanism, it would certainly allow you to do this in a much cleaner fashion. A simple example of this would be:

```
templateT, std::size_t N>

struct SizedThing

{

T data[N];

Count() {

return N; // sizeof(data) / sizeof(*data);

}

};

struct PreAllocation {
```

```
const bool HasNumber{ false };

const int Number{};

{};

constexpr int& number) : Number(number) {};

};

templateT, PreAllocation maybe>


{


std::vector data{};

AnotherSizedThing() {

{

data.reserve(maybe.Number);

  }

}
```

```
};

{



static void LookAtNonTypeParameters() {

std::cout << std::endl << std::endl << __FUNCTION__ << std::endl;

SizedThingmyNumbers;

std::cout << "Number of values in sized thing = " << myNumbers.Count() <<
std::endl;

AnotherSizedThingPreAllocation{} > maybeSomeValues;

std::cout << "Capacity in non-preallocated other thing = " <<
maybeSomeValues.data.capacity() << std::endl;

10 };

AnotherSizedThingpa > someValues;

std::cout << "Capacity in preallocated other thing = " <<
someValues.data.capacity() << std::endl;

}
```

```
};
```

We first create a providing the type to use, and the number we want in the collection. We create that for comparison to what we do next. We then create a couple of AnotherSizedThing objects. For the first we do not pass a value, but for the second one, we create a constexpr object and pass that in. That reserves space for values. The output is shown in the following Figure

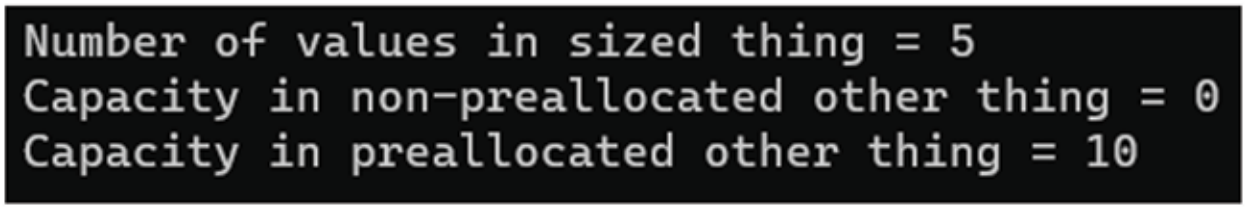

Figure 19.2: Output for LookAtNonTypeParameters

Recipe 19.4: Multidimensional subscript operator

Problem: We want a clean way of accessing multi-dimensional arrays.

Solution: That is a new feature in C++23.

For years we have been making workarounds and writing messy code so that we can have containers with multiple dimensions. The C++23 release now has an operator[] that accepts multiple parameters, allowing us to have multiple-dimensional arrays cleanly and easily.

Note that this is not the same as the new std::mdspan class from C++23. The std::mdspan class is a virtual view of data, where you specify the dimensions when you declare it, but can have multiple views (each with different dimensions) to the same set of data:

```
#ifdef __cpp_multidimensional_subscript

template T,

{

T const & x, y) const {

* MaxX + x];
```

```
}

T& x, y) {

* MaxX + x];

}

MaxX * MaxY> data;

};

template T, T NotFoundValue, MaxX, MaxY>


{

T& x, y) {

return data[x][y]; // map will create an entry

}

T x, y, int64_t z) const {

auto xi = data.find(x);

if (xi == data.end())
```

=

if ==

}

T>> data;

};

#endif

{

{

<< << << __FUNCTION__ <<

#ifdef __cpp_multidimensional_subscript

data;

// would give an out of bounds exception

```
// data[30, 30] = -1;

=

<< "value @ [3,7] = " << <<



=

NotFound, cdata;

=

auto res =

#else

<< "Feature not yet implemented for this compiler." <<

#endif

}

};
```

We also defined to give you an idea about the flexibility of this feature. We can tell the template a default value, and if we try to access a cell that

has not been set, then we get that default value back. This feature is not a true multidimensional array, since the operator[] overload calculates the net index, but it acts as it should, giving us the proper result. Refer to Figure

Figure 19.3: Output from LookAtMultidimensional

Recipe 19.5: std::make_shared supports arrays

Problem: We want a clean way of having a smart pointer for an array.

Solution: C++ 20 allows us to safer share arrays.

This shows what a great release C++20 was. Aside from the big four set of features, dozens of smaller features could make a big impact on how you coded. Now, instead of doing a complicated workaround by hand, or resorting to a third-party library, you can call to create a smart pointer of an array. Hopefully your mindset these days is to use make_shared and make_unique as much as possible, and only use new and delete in rare circumstances. Now, there is one less reason why you should risk a memory leak. Look at the following code:

```
{

int x;

};

{


{
```

```
<< << << __FUNCTION__ <<

=

=

<< "value1[37] = " << <<

<< "value1[99] = " << <<

auto arrMS2 =


  =

<< "value2[37] = " << <<

auto arrMS3 =

  =

<< "value3[42] = " << <<

=

=

// does not compile
```

```
// auto arrMU2 = std::make_unique();

}

};
```

This shows us allocating and initializing an integer array. You can use it as you would a normal array, but we did not have to use new or delete, and memory gets freed up automatically. It will also make resource ownership clearer, since you will not have the debate on who should delete the resource. We also create a shared array in our structure. With std::make_shared() in C++23, we can also now pass the size as a parameter. Of course the sibling function to std::make_shared is and we can now pass the size as a parameter for that as well. While you can create a unique pointer to some array with a parameter value of 100, we cannot create an array of 100 elements. Refer to Figure

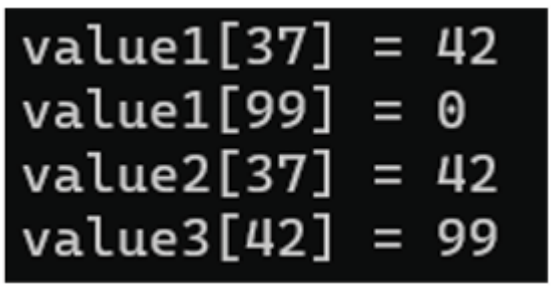

Figure 19.4: Output from LookAtMakeShared

In the output, we show that when we try to retrieve a value that was not set, we get a default value. There are a couple of related C++ features, but

we will save talking about those for the upcoming chapter on performance.

Recipe 19.6: CTAD improvements

Problem: We want a cleaner way of doing template argument deductions.

Solution: C++20 has made things easier, and better looking.

In C++17, there was a feature called Class Template Argument Deduction which meant you did not always have to specify the types when declaring an instance of your template. This was good in some cases but failed in others. Often, it worked fine with many standard library types but failed with your custom types that had initializers. The bad news was that in C++17, you had to create a workaround and create a deduction guide. This was to aid the compiler in what you were trying to do. The good news is that with C++20, you probably do not need the deduction guide any longer. This means a bit less code, and a cleaner way of doing things. Look at the sample code:

```
template T1, typename T2>

{

T1 first{};

T2 second{};

const =
```

```
};

// deduction guide was needed in C++17, but no longer

//template T1, typename T2>

//MyPair(T1,T2) -> MyPairT2>;

{



{



<< << << __FUNCTION__ <<

  {

pr1{ 2.3f };

pr2{ 2.3f };

if (pr1 == pr2) {

<< "pair1 {" << pr1.first << "," << pr1.second << "} =="
```

```
    << " pair2 {" << pr2.first << "," << pr2.second <<

  }

else {

<< "pair1 {" << pr1.first << "," << pr1.second << "} !="

    << " pair2 {" << pr2.first << "," << pr2.second <<

  }

<<

  }

  {

mpr1{ 2.3f };

MyPair 2.3f };

if (mpr1 == mpr2) {

<< "mypair1 {" << mpr1.first << "," << mpr1.second << "} =="

    << " mypair2 {" << mpr2.first << "," << mpr2.second <<
```

```
  }

else {


<< "mypair1 {" << mpr1.first << "," << mpr1.second << "} !="

   << " mypair2 {" << mpr2.first << "," << mpr2.second <<

}

  }

}

};
```

The first block shows the original intent for CTAD. Instead of being forced to explicitly state it was pair with template parameters, you can omit that part and the compile will deduce what you want. That worked well, although in some cases, it was finicky with the values. The problem arose when you had your own type. If you look at the MyPair struct, it has default initializers and that confused the compiler. The workaround was to have that odd looking (T1,T2) -> syntax. As mentioned, the newer compilers know better, and that part can be avoided. There is also a feature planned for C++23 that improves CTAD for inherited constructors. Although at this current time, none of the major compilers have completely implemented that feature.

Recipe 19.7: Deducing this

Problem: We want a cleaner way of writing functions that understand whether they should be returning const or not.

Solution: C++23 has a new mechanism that will cut down on the amount of code you should be writing.

The birth of this feature came about when some developers were working on other features, but both had the same issue: they were writing too many functions. (see references) To be thorough, they were writing different functions that did the same thing, but:

One was for non-const lvalues

One was for const lvalues

One was for non-const rvalues

One was for const rvalues

There could be others as well, such as if you were working with volatile codes. Most coders do not write all four versions for every function; some might write a const and non-const versions for some of their functions. If you are working on a library, the move coverage you have for different

situations would be better. Dealing with all the cvref (const-volatile-reference) situations can be burdensome. Being frustrated with all of this duplication, they worked on a mechanism that would cut down on all of that. If we think about it, when we call one-member function from another, we are implicitly passing so that it can be used in the called function. Instead of passing this implicitly, we can call it explicitly. We can also pass it with the reference operator, so it is an r-value reference. This is not required for all cases of all functions, but when you want to adhere to the or what you are dealing with or are working with large objects that you want to avoid copying, then this should certainly help.

This feature can be applied to many situations. We talked about Curiously Recurring Template Pattern in Chapter Views and Range With our deducing we can (in many places) remove some of that messiness, because we can forward to ourselves and it will know the cvref to use. Another case is for builder classes, where you want to chain together function calls. With typical CRTP, you would need a lot of extra calls to check what version of the builder you are working on, so that you can respond with the right object in order for the next builder function to work properly. Another use is that you can pass by-value of this for small objects, such as references)

If we look at the following code example:

```
#if || (_MSC_VER)

{

auto func1(this const Base& {
```

```
std::cout << "calling" << << "::func1" << std::endl;

}


auto func2(this {

std::cout << "calling" << << "::func2" << std::endl;

}

};


: Base {};

template ContainerValueType, std::size_t MaxX, std::size_t MaxY>

{


auto& at(this std::size_t x, y) {

* MaxX + x);

}
```

```
std::arrayMaxX* MaxY> data;

};

// passing this by value

T1, typename T2>

LessThanFn, const T1& lhs, const T2& rhs) {

< rhs;

}

};

#endif

class Chapter19_7 {

public:

static void LookAtDeducingSelf()

{

std::cout << std::endl << std::endl << __FUNCTION__ << std::endl;
```

```
#if || (_MSC_VER)


  {

  Base b1{};




  Derived d1{};




  }

  {

  Container2Ddata;

=

std::cout << "retrieved value =" << << std::endl << std::endl;

  }
```

```
  {

std::cout << "4 < 5 = " << std::boolalpha << << std::endl;

  }

#else

std::cout << "Feature not yet implemented for this compiler." << std::endl;

#endif

}

};
```

And just of note, the odd preprocessor check is due to, at the time of writing, the feature check __cpp_explicit_this_parameter was not defined, but it was working with the Microsoft compiler.

Our first code block shows how simple this feature is. When we use a derived instance, this knows which class it is. The second example shows how we can use and it knows whether it is getting or setting the value. The third code block shows that this can even be used by value.

This is certainly a new feature that others find new uses and patterns for, in the future.

Another point to mention is about a related feature. It can be used when you are using the deducing this feature, or you can use it elsewhere. We use std::forward() quite a bit with this sort of code, and in C++23, there is a new function called which applies the same properties to the type it is looking at. Whether it is const, or a lvalue reference type, it sorts out what the return type should be, based on the type provided.

Recipe 19.8: Transforming auto

Problem: We want a cleaner way of transforming a variable or value into another type.

Solution: In C++23, we can now use the auto keyword to do this.

This is another feature that will not make a coder's jaw drop in awe, but a nice little feature that will make our code cleaner. We have been told for a number of years to not do C-style casting. Aside from being a run-time risk, it can also cause some compiling issues. The compile will try the various casts, in the following order to see if they satisfy the requirements: (see references)

const_cast<>()

static_cast<>()

and then const_cast<>()

reinterpret_cast<>()

and then const_cast<>()

As mentioned, it will use the first one that satisfies the requirements. It may still cause the compilation to fail though.

This feature says that although you can create a new value by using the target type, such as in the following code when we have the cleaner method is to use auto, which gets the compiler to create the desired type. You can use the parentheses () or braces

```
{



n) {

<< << " value=" << n <<



}

{

<< << << __FUNCTION__ <<

1.2 };

// SampleFn(d); // compiles, but gives warning

// create new integer

// C-style cast
```

```
#ifndef _MSC_VER


d });

#endif

}

};
```

As you can see in Figure although we are passing in a the auto cleanly converts it to an integer. The i as the type (instead of looks odd, but is an accepted quirk of some compilers.

```
LookAtAuto
i value=1
i value=1
i value=1
i value=1
```

Figure 19.5: Output from LookAtAuto

Recipe 19.9: char8_t and std::u8string types

Problem: We want a clean type when using characters from UTF-8 strings.

Solution: C++20 added the type char8_t and the u8string type.

For better or worse, we are all used to dealing with strings of different encodings. The UTF-8 strings have been around since C++11, but until recently, we did not have a type to deal with the individual characters.

```
{



{

<< << << __FUNCTION__ <<

#ifdef __cpp_char8_t

char8_t c8 =

u8str = u8"Hello

#endif
```

```
u16str = u"Hello

u32str = U"Hello

widestr = L"Hello

u8str2{ u8str };

}

};
```

Moreover, in the C++20, we have which is defined as This opens the door to all of the string functionality, for the UTF-8 encoding.

There is also std::pmr::u8string added. The pmr namespace was added in C++17, but not talked about much, and so we will touch on that in the upcoming chapter about making code faster.

Recipe 19.10: Suffix for std::size_t

Problem: We want a clean way of setting a value to

Solution: C++23 has added a new suffix uz or

We will switch from prefixes in the last topic, to suffixes in this topic. Another feature that is not big, but will make your code neater. We now have a z suffix, that will identify a value as a You would use uz for unsigned, and z for signed. The casing and order (of is said to not matter, but you should be consistent with the rest of your code-base.

```
{
```

```
{
```

```
<< << << __FUNCTION__ <<
```

```
#ifdef __cpp_size_t_suffix
```

```
= // or UZ, or Uz, or uZ, or ...
```

```
= // or Z
```

```
#endif

}

};
```

The suffixes should also work with std::ptrdiff_t values.

Recipe 19.11: Named universal character escapes

Problem: We want a clean way of using universal character escapes.

Solution: This was added to the C++ 23 release.

Aside from the traditional escape sequences for things such as and \ (among others), there is also a large set of named escapes. These may not be widely known, but use of this Unicode standard has existed in other languages for a while, so it is good that they are finally in C++23. (see references) To use them, you have and then put the escape character name within braces.

```
{



{

<< << << __FUNCTION__ <<

#if >= 202207L

<< "\N{OHM SIGN},\N{LATIN CAPITAL LETTER A},"

  << "\N{LATIN CAPITAL LETTER A WITH MACRON}" <<
```

```
#endif

}

};
```

While it means you might have to look up a lot of names, the end result in the code is far cleaner and more understandable than numerical Unicode values. The output for this, in Figure is short but sweet.

Figure 19.6: Output from LookAtNamedEscapes

Recipe 19.12: Monadic operations for std::optional

Problem: We want a clean way of chaining functions together.

Solution: C++23 offers a mechanism of having functions return and chain them together.

The std::optional mechanism was added in C++17. Some people like as a handy way of returning the result, or being clear that you do not have a result. This removes the requirement of extra Booleans or other variables saying whether the code was successful. Some like it, some do not, and some still do not know about it. For the latter, we will do a little recap: (See references)

```
{

S&) =

S(S&&) =

S() =

};

{

EasyS() {}
```

```
x) : X{ x } {}


};

{


part1,

part2 = {}, part3 = {

<< part1 << " " << << " " << part3 <<


}


{

<< << << __FUNCTION__ <<

  {

o;

// compiler error
```

```
// o = S{}; // cannot do assignment

// can create a default new S inside of optional, since cannot move or copy

o.emplace();

auto es2 = EasyS{};

auto o2 =

auto o3 =

es3 =

es4{ 37 };

<< "es3=" << es3.value().X << ", es4=" << (*es4).X << <<

  }

  {




{},

  }
```

<<

}

};

The templated class was added so you could check if an object had a value. We have created a couple of simple structs. For the first struct S, we took away some operations such as a copy constructor. The second struct EasyS has 2 constructors, one of which initializes an integer member. In our LookAtOptional_Recap() function, we create an and to show one "trick", we cannot do an assignment, but we can emplace another S within our optional.

We then work with the EasyS struct, and show how we can call We can pass it an existing struct, or we can pass a value that is a parameter for the struct's constructor. We also showed that we can create an optional by using parentheses and passing in a value for the struct's constructor, after we specify The last thing we do is print out the struct's value; either by using optional's value() member function, or by de-referencing the optional variable. De-referencing will help if you want to transition existing code that uses a pointer to access the members. Hopefully, you do not have to change any code at all.

We then look at something different, instead of creating an optional that we might return from a function, we can also use when calling a functional. Instead of having multiple default values, you can use an optional instead. This also could help if you do not want to expose what a default value would be for a function, keeping that implementation private. For our

PrintMessage() function, the second parameter is optional, with a default/empty value, and the last parameter is a traditional defaulted variable. Passing 3 values is clear enough and passing the first 2 values just means the third parameter uses the default provided. If we only provide 1 value, then we can see in the implementation code it uses the value_or() function to provide a value. In the last call to we can see that the optional allows us to skip the second parameter. The output is shown in Figure

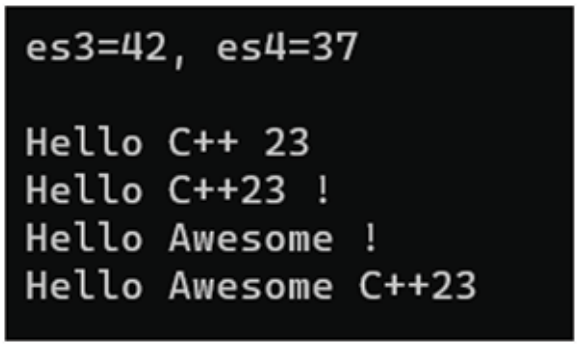

Figure 19.7: Output from LookAtOptional_Recap

In C++23, they wanted to build off of that, to make the usage flow better and more functional. They wanted to be able to chain callables together somehow. Either with a "this or that" thinking, or a "this, then that". The functions that were added were:

This should allow callables (functions, or lambdas) to be called in sequence to enhance the end result. If one of the links results in returning an empty or then it will know to see the processing.

This allows you to change the type of templated parameter of your optional. The last callable must match the expected type, but if you had a process that transforms an object, then this will be a good choice. If, along the chain of

transformations, the result of a callable is an optional that does not have a value, then the chain is stopped.

value_or: This returns the contained value if it exists, and if not then it will use the second value given.

This function would define the callable if you had an empty optional and wanted to provide a fallback value.

The following code sample is a bit long, but it gets back into the food theme. We have a builder class, and provide a list of ingredients that we have. We can ask it to build something and check what is required, and this either results in dinner, or an empty optional. We have that for a few dishes, and then we have a MakeSomething() function that tries to make anything, based on what ingredients we have. If all else fails and we do not have the ingredients, then it just orders for some food.

{

Dough,

Pasta,

Lettuce,

Tomato,

Onion,

Cheese,

Sausage,

Sauce,

Garlic,

Mushroom,

};

name; };

{

ingredients) :

MyIngredients(ingredients) {

}

{

Ingredient;

result;

```
if Dough) &&

Sauce) &&

Cheese)) {

  result = Dinner{ "Pizza" };

AddOtherIngredients(result);

  }

else {

  result =

  }

return result;

};

{

Ingredient;

result;
```

```
if Pasta) &&


Sauce)) {

  result = Dinner{ "Pasta" };

AddOtherIngredients(result);

  }

else {

  result =

  }

return result;

};

{

Ingredient;

result;

if Lettuce)) {
```

```
  result = Dinner{ "Salad" };
AddOtherIngredients(result);
  }
else {
  result =
  }
return result;
};
Dinner { return Dinner{ "takeout" }; };
Dinner {
dinnerInProgress = MakePizza();
if == {
dinnerInProgress = MakePasta();
  }
```

```
if == {

dinnerInProgress = MakeSalad();

  }

? dinnerInProgress.value() : OrderDinner();

}

Dinner {

return

}


MyIngredients;

odinner) {

  Dinner& dinner = odinner.value();

toppings;

Ingredient;

for : MyIngredients) {
```

```
switch {

case Dough:

case Pasta:

case Lettuce:



case Tomato: toppings += "tomato,

case Onion: toppings += "onion,

case Cheese: toppings += "cheese,

case Sausage: toppings += "sausage,

case Sauce: toppings += "sauce,

case Garlic: toppings += "garlic,



case Mushroom: toppings += "mushroom,

  }

  }
```

```
  dinner.name += ? " : plain" : ", with: " + toppings);

}

};

{

{



<< << << __FUNCTION__ <<

#ifdef __cpp_lib_optional

DinnerMaker> dinners{

{ DinnerMaker({Ingredient::Lettuce,

Ingredient::Mushroom, Ingredient::Tomato }) },

{ DinnerMaker({Ingredient::Dough, Ingredient::Cheese,

Ingredient::Mushroom, Ingredient::Tomato, Ingredient::Sauce }) },

{ DinnerMaker({Ingredient::Pasta, Ingredient::Sauce,
```

```
Ingredient::Mushroom, Ingredient::Tomato }) },

{ DinnerMaker({Ingredient::Sauce,

Ingredient::Mushroom, Ingredient::Tomato }) },

  };


for DinnerMaker> dinnerPair : dinners) {

auto d1 = dinnerPair.second.MakePizza();

<< "From trying to make pizza " << dinnerPair.first << " we have : "

    << ? d1.value().name : <<

auto d2 = dinnerPair.second.MakePasta();

<< "From trying to make pasta " << dinnerPair.first << " we have: "

    << ? d2.value().name : <<

auto d3 = dinnerPair.second.MakeSalad();

<< "From trying to make pasta " << dinnerPair.first << " we have: "

    << ? <<
```

```
auto d4 = dinnerPair.second.MakeSomething();

<< "From trying to make something " << dinnerPair.first << " we have: " <<
d4.name <<

<<

  }

<<

#endif

}

};
```

We create a map, pretending we have different ingredients available on different days. The output is shown in Figure

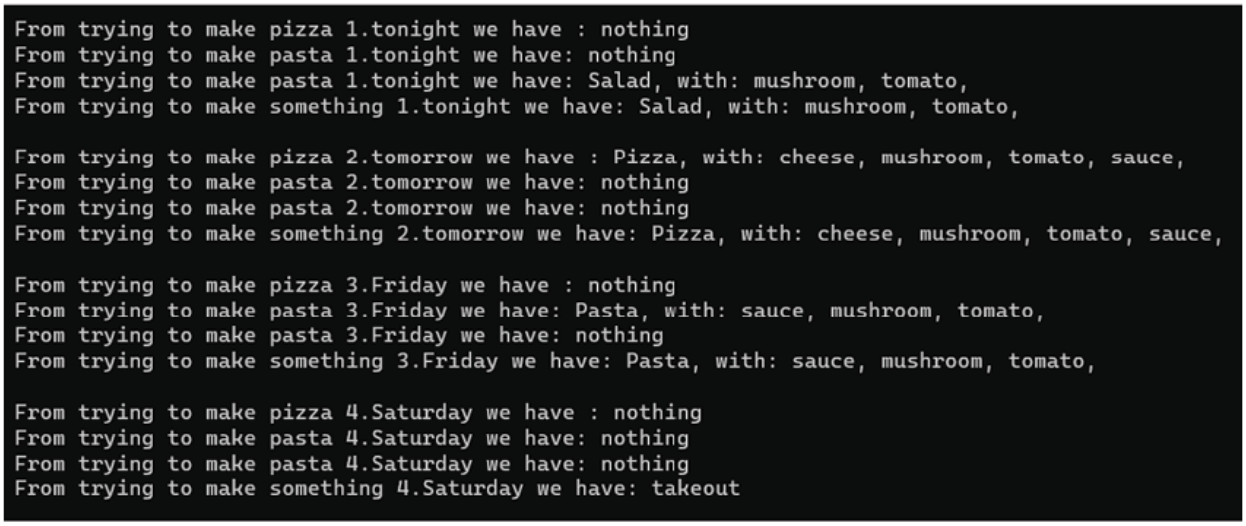
From trying to make pizza 1.tonight we have : nothing
From trying to make pasta 1.tonight we have: nothing
From trying to make pasta 1.tonight we have: Salad, with: mushroom, tomato,
From trying to make something 1.tonight we have: Salad, with: mushroom, tomato,
From trying to make pizza 2.tomorrow we have : Pizza, with: cheese, mushroom, tomato, sauce,
From trying to make pasta 2.tomorrow we have: nothing
From trying to make pasta 2.tomorrow we have: nothing
From trying to make something 2.tomorrow we have: Pizza, with: cheese, mushroom, tomato, sauce,
From trying to make pizza 3.Friday we have : nothing
From trying to make pasta 3.Friday we have: Pasta, with: sauce, mushroom, tomato,
From trying to make pasta 3.Friday we have: nothing
From trying to make something 3.Friday we have: Pasta, with: sauce, mushroom, tomato,
From trying to make pizza 4.Saturday we have : nothing
From trying to make pasta 4.Saturday we have: nothing
From trying to make pasta 4.Saturday we have: nothing
From trying to make something 4.Saturday we have: takeout

Figure 19.8: Output from LookAtOptional

Recipe 19.13: Using std::expected

Problem: We want a clean way of returning a value, or an error.

Solution: In addition to the std::optional changes, C++23 also has a std::expected template.

It makes sense that while std::optional nicely allows us to return a value, or not, it does ask the question on Why did we not get a That led to std::expected being created, which holds either the expected value or an error value. Since it is templated, your error value can be anything you want, an integer, an enum, an exception, or in our case, a string:

```
{



inline BigProblem = "Kitchen is in

#ifdef __cpp_lib_expected

std::string> MakeDinner() {

DinnerOutcomes{

"Missing "Appliance Not BigProblem,

  };
```

```
std::ranges::rotate(DinnerOutcomes, DinnerOutcomes.begin() +

std::string currentOutcome =

if == {

"Something delicious" };

  }

else

  {

  }

}

#endif

static void LookAtExpected() {

std::cout << std::endl << std::endl << __FUNCTION__ << std::endl;
```

```
#ifdef __cpp_lib_expected

for (int i = i < i++) {

  auto dinner = MakeDinner();

if {

std::cout << "Tonight we are eating " << dinner.value().name << std::endl;

  }

== Chapter19_13::BigProblem) {

std::cout << "We will not be cooking anything for a while because: " <<
dinner.error() << std::endl;

  }

else {

std::cout << "Dinner will be delayed because: " << dinner.error() <<
std::endl;

  }

  }

#else
```

```
std::cout << "Feature not yet implemented for this compiler." << std::endl;


#endif

}

};
```

We are planning to make dinner, and we have a list of reasons why dinner failed. For the sake of simplicity, we have a blank entry in our list of errors, to indicate there was success. With the help of a static variable, and a range function, we rotate through our outcomes. If the current outcome is good (meaning our current error is empty) then our return value is a nice If we have a problem, then we return a with the following outcome:

```
Dinner will be delayed because: Missing Ingredient
Dinner will be delayed because: Appliance Not Working
Dinner will be delayed because: Burnt
We will not be cooking anything for a while because: Kitchen is in ashes
Tonight we are eating Something delicious
Dinner will be delayed because: Missing Ingredient
Dinner will be delayed because: Appliance Not Working
```

Figure 19.9: Output from LookAtExpected

If all goes well, we can get our result, as we did with with If not, then we can look at the error result via the error() function.

The std::expected class has a number of functions that exist in the std::optional class. We have used has_value() and value(), but there is also value_or() that we previously used. We also should have the monadic operations, such as A new monadic function is added to which is This function will return the expected value if it has it, otherwise will try to right a wrong and fix the error. While std::optional still had doubters as a return type, many are liking the idea that this helps in error handling. The consumer does not have to be always watchful for exceptions, and just check if they got a successful response, or can see what error they received.

Just note that if you are investigating what the best path for your situation is, there is another idea to consider. The std::variant templated class from C++17 is something to consider. This can hold one (and only one) value, from the various types that you said it might be. You would check what type is currently being used with the std::holds_alternative() function.

Recipe 19.14: Pre-processing directives elifdef and elifndef

Problem: We want a cleaner way of checking for definitions when pre-processing.

Solution: In the C++23 release, they added and

Nobody enjoys going through the #ifdef logic when looking at code. It makes it a less enjoyable job when you need to look at big nested blocks, trying to sort out the logic. In C++23, they added two more directives; #elifdef (which is equivalent to #elif and #elifndef (which is equivalent to #elif ! Although not essential, the new directives make code more readable.

```
#define ThisIsDefined 123

{



{

<< << << __FUNCTION__ <<

#ifndef ThisIsDefined

<< "Did not expect to be here!" <<
```

```
#elifdef DefinitelyNotDefined

<< "Should not have this defined!" <<

#elifdef AlsoDefinitelyNotDefined

<< "Should not have this defined either!" <<

#else



<< "Expected to be here!" <<

#endif

#ifdef DefinitelyNotDefined

<< "Should not have this defined!" <<

#elifdef AlsoDefinitelyNotDefined

<< "Should not have this defined either!" <<

#elifndef ThisIsDefined

<< "Expected to be here!" <<
```

```
#else
<< "Did not expect to be here!" <<
#endif
}
};
```

Although this new feature can be helpful, there is an issue to look out for. It does not seem that this feature has a feature-test macro for it, and not all compilers have implemented the feature yet. Which causes the problem of how do you use the compiler preprocessor if there is no way to check if it exists? When we run our sample code on the Microsoft gcc or clang, we get the output as shown in Figure The Microsoft compiler only gave some of the correct output by chance. It does not seem to know about the so it jumped down to the

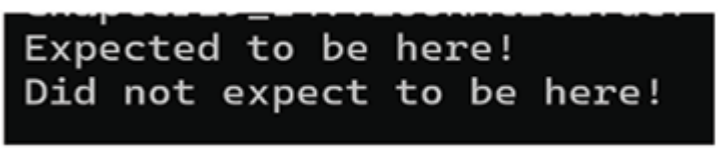

Figure 19.10: Output from LookAtElifdef using MSVC

The other compilers state they have completed the feature, but they are not getting the correct result either, as shown in Figure

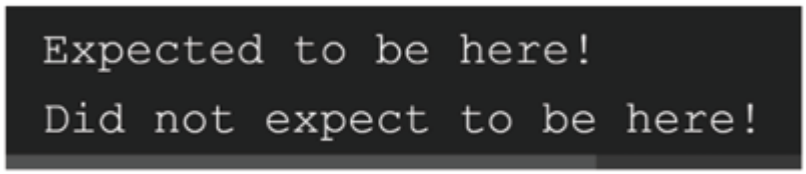

Figure 19.11: Output from LookAtElifdef using gcc and clang

Recipe 19.15: invoke_r

Problem: We want an easy way getting a different return value from invoking a callable.

Solution: C++23 added

This is another simple feature that makes our code a little bit cleaner. Instead of our code checking and converting a return value, C++23 added a library call for us to use. The function std::invoke_r() acts in the same manner as but implicitly converts the return value (unless void is specified). It just helps clean up our code a little bit, one less cast in our code makes it slightly more readable.

```
{



{

<< << << __FUNCTION__ <<

auto PrintSomething = x) -> void { << x << };

auto Doubler = x) { return x * };

auto result =
```

```
<< "Value of " << << " type = " << result <<


}

};
```

Our coding example is straightforward. We call a function that returns an int, but we will be using it as a and invoke_r<>() does that nicely. Although there is not any extra value, we use instead of the normal to call a function that does not have anything returned. Since we specify to convert it to void, it does not return anything.

Recipe 19.16:

Problem: We want a cleaner way of dealing with input/output for span buffers.

Solution: In C++23, there is a new header that defines that functionality.

While we have std::istringstream and std::ostringstring to work with strings and input/output, it is often cleaner when you do not have to think as much about string details. The abilities given to us through the header in a way, allow us to handle it in multiple ways. Now that we are using spans a lot more, it would be nice to deal with streaming into, or out of a span of characters. The simple code example here shows that we do the input into, or out of, a character span:

```
{



{

<<

<< << << __FUNCTION__ <<

  {

= "37 42
```

```
is{ };

int n1, n2, n3;

  is >> n1;

assert(n1 ==

  is >> n2;

assert(n2 ==

  is >> n3;

assert(n3 ==


<< "From" <<

    << ", n1=" << n1 << ", n2=" << n2 << ", n3=" << n3 << <<

  }

  {


};
```

```
int 37 }, n2{ 42 }, n3{ 67 };

os << n1 << n2 << n3;

= os.span();

  ==

sp.size()) ==

auto x1 = sp.data();

auto x2 =

assert(x1 == x1);

==

<< "From n1=" << n1 << ", n2=" << n2 << ", n3=" << n3

    << ", we have output of: " << << <<

  }

}

};
```

The results are shown in Figure

```
From"37 42 67", n1=37, n2=42, n3=67

From n1=37, n2=42, n3=67, we have output of: "374267"
```

Figure 19.12: Output from LookAtSpanStream

Recipe 19.17: Bit initialization

Problem: We want a clean way of working with bits, and to initialize their value.

Solution: In C++20, they added an initializer.

This mechanism for having a bitfield has been around for a number of releases. It can be handy to say how many bits a variable will take, and you can pad it by not specifying a variable name. The recent release added the ability to initialize the bit-field. The following example does not use all the fields we have defined, but it shows how we initialized the bits3 and bits16 fields:

```
{


: // blocked against usage


: // force to start a new byte

=
```

```
};

{



{

<< << << __FUNCTION__ <<

for = i <= i *= {

BitField i - 2 };

<< "val=" << << ", bits=" << <<




<< "val=" << << ", bits=" << <<

  }

}

};
1.
```

Recipe 19.18: Extend init-statement to allow alias-declaration

Problem: We want an easy way of having a using statement in some code blocks.

Solution: C++23 allows a using statement in a for loop.

This little feature was perhaps more for consistency, rather than necessity. Since we can have using statements in other parts of the code, people requested it be available in coding blocks as well. This will lessen the usage of typedef statements. Although originally wanted in if and switch statements as well, for C++23 we have them in range-based for statements:

```
{



{

<< << << __FUNCTION__ <<

};

// old way of defining a type

for T; T n : numbers) {
```

```
  }

// range-based for loop from C++11

for : numbers) {

  }

#if >= 200704L

using Num =

for (Num n : numbers) {

  }

#else


<< "Feature not yet implemented for this compiler." <<

#endif

}

};
```

Conclusion

We have had yet another chapter where we have talked about a lot of features that you can use. The theme of this chapter was to show that it is not always the big flashy features that will make your code a lot better. Using some of the techniques mentioned here should make your code cleaner. That means that it is easier to read, easier to understand, easier to debug, and easier to expand upon. It might be some new prefixes or suffixes that you think make the code easier to understand. You might see that deducing-self mechanism will cut a ton of duplication out of your library code. It might also mean you shift to std::expected and have a whole new paradigm on the return values you give back. Some of these are big sweeping changes, some are small, but any and all should make your code cleaner.

Points to remember

There are times when we do not always want to use the auto keyword. Providing types to your lambdas might cut out mistakes.

Having the explicit object parameter, otherwise known as deducing or deducing self will cut down on a lot of extra code and simplify CRTP.

Using multidimensional syntax, or make_shared for arrays, will help code that handles your collections look smoother.

Switching to and/or may be a big change, but there are benefits that you should consider.

C++20's lambda parameter packs and familiar template syntax are nice improvements for lambdas.

Using the new features with span is handy.

References

YouTube. (n.d.). FTSIILES. Retrieved from https://www.youtube.com/watch?v=uOc6RPu9-CA

MicrosoftDevBlogs. (n.d.) DeducingSelf. Retrieved from https://devblogs.microsoft.com/cppblog/cpp23-deducing-this/

YouTube. (n.d.) UsageOfThis. Retrieved from https://www.youtube.com/watch?v=jXf--bazhJw

CppReference. (n.d.) AutoOverCStyle. Retrieved from https://en.cppreference.com/w/cpp/language/explicit_cast#Explanation

Unicode. (n.d.) UnicodeEscapes. Retrieved from https://www.unicode.org/Public/14.0.0/ucd/NamesList.txt

YouTube. (n.d.) OptionalRecap. Retrieved from https://www.youtube.com/watch?v=iah8giAKvUM

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 20

Making Your Code Safer

Introduction

As we have touched on before on various topics, recent releases have introduced new keywords, classes, and constructs improve the safety of your code. Some of these would include all of the new concepts that were created (or the ones you can create) to restrict or better define template parameters. They could relate to all of the new const-related keywords which trigger compiler errors versus having issues at runtime. Certainly, safety was paramount with std::jthread when it automatically joins threads so it does not crash like Using a mechanism such as assuming a function will return a proper value or an error value, should make the code a lot safer. Of course, there are many more that we have mentioned, and this chapter will talk about a few new additions as well.

Structure

In this chapter, we will cover the following topics:

Recipe 20.1: Fixed width floating-point types

Recipe 20.2: Conditionally explicit constructor

Recipe 20.3: #warning

Recipe 20.4: std::start_lifetime_as

Recipe 20.5: std::out_ptr(), std::inout_ptr()

Recipe 20.6: Overloads of std::to_chars() and std::from_chars()

Recipe 20.7: constexpr std::bitset

Recipe 20.8: std::to_underlying

Recipe 20.9: Direct initialization

For these features, we have most of them working with the various compilers. Unfortunately, the least implemented feature is the std::start_lifetime_as (and related functions), but otherwise, it is positive.

Apologies for the maybe with as the example uses std::expected. That feature was supposed to be working by now with all compilers, but at the time of writing the clang compiler was having issues with it.

}

Objectives

By the end of this chapter, the reader will understand some new ways to make their code safer. To re-emphasize, this chapter is probably shorter than others, not because safety was not a consideration in recent C++ releases, but because we have already talked about some features in previous chapters that deal with creating safer code. By now, you should understand how all of those other new features previously mentioned can help you.

Recipe 20.1: Fixed width floating-point types

Problem: We want a safer way of working with floats.

Solution: C++ 23 has a number of new fixed width floating-point types.

The latest C++ release has a new header named This has many new types defined, to help ensure you are using the right size for specific situations. They are all in the std namespace, and each has its literal suffix (that works in both lowercase and uppercase. The types are listed in Table

Table 20.1: Types under

The std::bloat16_t is known as the Brain Floating Point (See references). It is used in a number of new processors to reduce storage and give faster estimate values, to improve the speed of machine learning algorithms. You can still use the existing types long but the new types are more explicit, and certainly, the new types might be better applicable in some situations. If there are cases with sending, saving, or interacting, and you need a certain precision, then you want to use the appropriate type.

Recipe 20.2: Conditionally explicit constructor

Problem: We want a safer way of how, or when, constructors are called.

Solution: C++ 20 has a new mechanism for the explicit specifier.

For some constructors, what started as a feature turned out to be a flaw. A converting constructors is where a type can be implicitly created by specifying a value. These constructors are the source of many issues over the years, where you inadvertently call a function with a value, not realizing that it creates another object.

Once again, let us mention the headers we will be using in this chapter:

```
#include

#include

#ifdef __cpp_lib_print

#include

#endif

#include
```

```
#include

#include

#include

#include

#include

#include

#include

#include

#include

#include

#include


#include

#include
```

```
#include

#include

#include

#include

#include

#include

#include

#include

#include

#ifdef __STDCPP_FLOAT16_T__

#include

#endif

#include

#include
```

```
#include

#include

#include
```

An example of this is in the following code:

```
{


int _v1;

_v2;


i1) : _v1(i1), _v2() {

// perform long or resource-laden operation

}


s1) : _v2(s1), _v1() {

// perform long or resource-laden operation
```

```
}

const { return _v1; }

};

{


int _v1;


DangerousThing2() =

: _v1(i1) {

// perform long or resource-laden operation

}

};

{


dt) {}
```

```
DangerousThing2& dt) {}

{
DangerousThing dt1 =
DangerousThing 2 };
DangerousThing dt3 =

/*does not compile, since cannot implicity convert from const char* to string,

  and then to DangerousThing */
// WorkOnDangerousThing("Hello");
//
// this works because explicitly creating a string, which can be used to implicitly create DangerousThing
// this works because only doing 1 implicit conversion from const char* to string
```

```
= 1 + dt1;

  WorkOnDangerousThing2({});

}

};
```

The example shows that we can call WorkOnDangerousThing by passing an integer, even though it says that the parameter is a That is because it can implicitly create a We can also call a function just by passing an empty initializer {

That is why the explicit keyword was initially created in C++11, removing the ability to instantiate a type unless explicitly saying that was the intent. The explicit keyword could be used on constructors and operator functions. That solved the main problem, aside from the problem that not enough developers are using the explicit keyword yet. (That is a subtle suggestion, but not the focus of this topic).

```
{


int _v1;

std::string _v2;
```

```
explicit =

explicit i1) : {}

explicit std::string& s1) : {}

explicit operator const { }

explicit operator const { return _v1; }

};

{


dt) {}


{

SaferThing st1 =

SaferThing 2 };

// these no longer works because cannot do implicitly
```

```
// SaferThing st3 = 3;

// WorkOnSaferThing(4);

//




if (ot1) {} // can be use because of operator bool

//bool flag3 = ot1; // can be used in an if-block as above, but not to capture the value

//const bool flag4 = ot1; // even const bool cannot capture value of operator function



// int oddOperation = 1 + ot1; // does not compile, since cannot add different types

= 1 +

}

};
```

The preceding code sample now shows the use of the explicit keyword. This restricts the code from calling a function with just an integer, or

adding different types when it is using operator This dramatically reduces the chances of calling a function with the wrong type, or not realizing we are creating an expensive object.

People found that some cases were too restrictive, especially with templated types. You might not want to implicitly create an object with a value of one type, but in other cases, it would be fine to call the constructor. In C++23, they added the ability to add an expression to the explicit keyword, which filtered when the constructor could be called. If we look at the following code sample:

```
template T>

{



SafeAndEasierThing() =

SafeAndEasierThing(T&& t) : { }

T& t) : value(t) {}

//explicit(sizeof(T) < 10) SafeAndEasierThing(const T& t) : value(t) {} // possible option for constructor

const { return value; }
```

```
T value;

};

{


a) {}

template T2>

SafeAndEasierThing a) { }


{

// does allow trivial creation of SafeAndEasierThing that can be passed to function

// WorkWithSafeAndEasierThing(DangerousThing{1}); // cannot create SafeAndEasierThing from non-trivial

  SafeAndEasierThing DangerousThing { 1 } };

WorkWithSafeAndEasierThing(st1);

}
```

```
};
```

This shows that we can still create an object indirectly with a trivial object, but if it is something more complex, then we need to do that explicitly. The expression you use can be varied from std::is_trivial_v() to a sizeof() restriction, to perhaps a concept that you think would be a safe restriction. (See references)

Recipe 20.3: #warning

We want an easy way of knowing there might be an issue.

C++ 23 has a new preprocessing command

We have to state that #warning is new, since it is now an official part of the language. Although, some compilers have had the #warnng statement for a number of years.

Code safety can come in many ways; whether that is type-checking, bounds-checking, parameter-checking, and many more. With all of the const-related features adding in C++20, that means that you can do a lot more processing at compile time, rather that runtime. One way to add extra safety, is to warn developers when there might be a problem. The #error directive has existed for a while, and now you can also use a #warning directive to warn coders (or at least automated build tools) that there is a potential problem. You may not want the build to break, but you do want to raise the issue that it should be looked into.

While on the topic of code safety, here are a few suggestions:

Have a goal for your builds to (eventually) be on the maximum build warning level. The more warnings you see, hopefully, that means that more of potential issues get looked at. The more issues that get looked at, hopefully more issues are found and get fixed. The more issues that get fixed, the safer your code will be.

Use different tools to find issues. Aside from your usual compile, use external tools that could find potential issues as well.

There are also sanitizers that can be used in test-builds, to try to find issues. This implies that you have more than the debug vs release builds. A release build is good for a staging environment, but if you have a testing environment, and maybe a performance environment, then perhaps you have a build that still does a little bit more than what a release build does. This is not the focus of the book, but there are tools out there that help harden what you deploy.

Although code coverage metrics have their naysayers, it is still some extra information that indicates how solid your code base is tested.

Here is a quick example of

```
#define CurrentLevel 123

#define 130

{



{

#ifndef _MSC_VER

< TargetLevel
```

```
#warning "Code may not build, or work, as hoped"

#endif

#endif

}

};
```

Although not working on the Microsoft compiler yet, we can show the output from clang in Figure

```
<source>:173:4: warning: "Code may not build, or work, as hoped" [-W#warnings]
  173 |                 #warning "Code may not build, or work, as hoped"
```

Figure 20.1: Output from LookAtWarning

Recipe 20.4: std::start_lifetime_as

Problem: In some situations, we want things to be safer by using explicit lifetime management.

Solution: C++ 23 has

This is another feature for specific use-cases. The new function informs the compiler to treat a range of memory as if it were a specific type. Coders often get into trouble when they make the assumption of what a block of memory is. Sometimes this is done with a simple C-style case, or sometimes it is done via That is called type punning, and can be very risky. (See references)

There are a few variations of this templated function:

returns a pointer to type T.

returns a pointer to const type T, if supplied const input. Similarly, if the input is volatile, or const volatile.

returns a pointer to type T, given a pointer and a size. There are similar overloads if the input is const, volatile, or const volatile.

If you are calling reinterpret_cast quite a bit, this is a safer, and recommended alternative.

Recipe 20.5: std::out_ptr(), std::inout_ptr()

Problem: We want a safer way of working with C APIs.

Solution: C++ 23 has added std::out_ptr and

Although this is very specific to working with C APIs, we just wanted to mention that this should allow better (and safer) usage. It creates a temporary pointer-to-pointer that updates the provided smart pointer when it destructs.

For an example, as it says in the proposal (See references), an out_ptr can be used to get a resource from a C API:

```
// C code in another file

handle);

struct resource_deleter {

handle) {

c_api_delete_handle(handle);

}
```

```
};

// C++ code

{



{

// C++ code

resource_deleter>

  auto errNumber = c_api_create_handle(

std::out_ptr(resource)

  );

}

};
```

We get the luxury, and safety, of using a resource with unique_ptr, or shared_ptr, but we can still communicate with older C code, and cleanly

consume a resource.

Recipe 20.6: Overloads of std::to_chars() and std::from_chars()

Problem: We want a safer way of converting a number to a string.

Solution: C++ 23 has a few different additions to std::to_chars() and

Although it is nice when we can just use a std::string to convert our number, we do not always have that luxury. Many times, we have to work within a fixed sized array of characters.

The utility function std::to_chars() has existed since C++17, but in C++23, it has had a few additions. Before we talk about the new features, since it is relatively new, we can do a quick recap:

```
{



args)

{

10> strBuffer;

result = strBuffer.data() + strBuffer.size(), args...);

if {
```

```
<< result.ptr)) <<

  }

else {

<< <<

  }


}


{

<< << << __FUNCTION__ <<

pi =


PrintOutChars(-pi);

PrintOutChars(-pi,

PrintOutChars(-pi,
```

}

};

We call std::to_chars with a character buffer and the input(s). We can specify just to value, or even some formatting characteristics, as seen in Figure

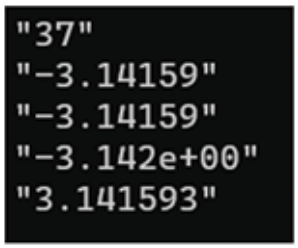

Figure 20.2: Output from LookAtToChars_recap

The first new feature to mention relates to it being constexpr for integral inputs. The returning value will be which will help with cases that can have compile-time checking.

{

{

<< << << __FUNCTION__ <<

```
10> strBuffer;

#if >= 202207L

result = strBuffer.data() + strBuffer.size(),

if {

<< "Result=" << result.ptr) <<

  }

else {

<< "Problem" << <<

}

#else

<< "Feature not yet implemented for this compiler." <<

#endif

}

};
```

The output being simplistic, but shows it working in Figure

Figure 20.3: Output from LookAtToChars

The next new feature is that the inputs are no longer just float, double, or long double. They should accommodate our new floating-point types of and

Of course, converting to a string is just half of the feature set. Similar changes were made to Those would be:

Ability to convert to the new floating-point types as well.

When converting to integers, the returning value can be

We have a quick recap of what the function does, and you should be able to easily apply the new features to the code given:

templateT>

std::expectedstd::string> std::string_view s) {

T value{};

std::from_chars_result result = std::from_chars(s.data(), s.data() + s.size(), value);

```
if (result.ec == std::errc{}) {

return value;



}

else {



}

}

{


static void LookAtFromChars() {

std::cout << std::endl << std::endl << __FUNCTION__ << std::endl;

  {

std::cout << "Looking at integers" <<

  auto intStrs = { "2023C++31" };
```

```
for (auto str {

#if >= 202207L

    auto result = ParseTextToNumber(str);

if {

std::cout << "For input" << std::quoted(str) << ", parsed value: " <<
result.value() << std::endl;

    }

else {

std::cout << "For input" << std::quoted(str) << ", got error: " << result.error()
<< std::endl;

    }

#else

std::cout << "Feature not yet implemented for this compiler." << std::endl;

#endif


  }
```

```
  }

  {

std::cout << "Looking at floats" <<

  auto floatStrs = { "2023C++31" };

for (auto str {

#if >= 202207L

    auto result = ParseTextToNumber(str);

if {

std::cout << "For input" << std::quoted(str) << ", parsed value: " <<
result.value() << std::endl;

    }

else {

std::cout << "For input" << std::quoted(str) << ", got error: " << result.error()
<< std::endl;

    }

#endif
```

```
  }

  }

}

};
```

Just to reuse some of the features we have learned about, we are returning a std::expected result. This allows us to easily capture the error text, otherwise, we can use the value returned in Figure

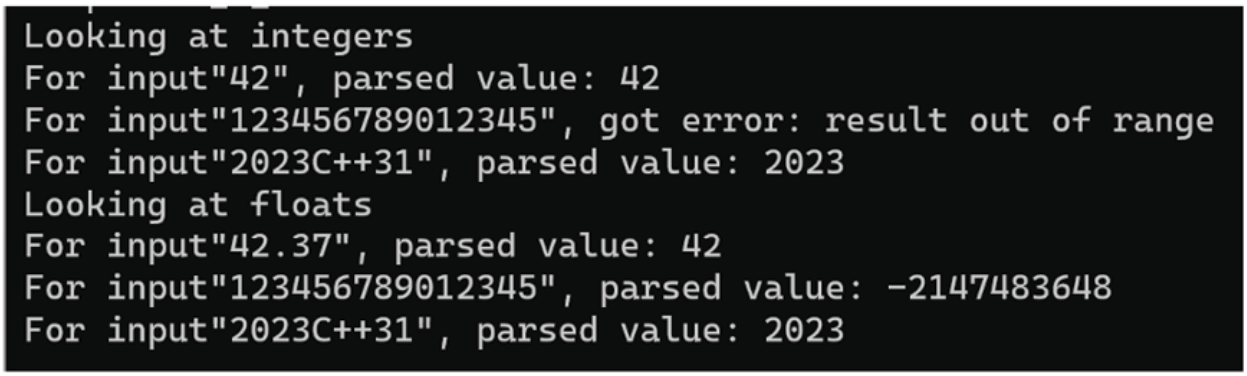
Looking at integers
For input"42", parsed value: 42
For input"1234567890123345", got error: result out of range
For input"2023C++31", parsed value: 2023
Looking at floats
For input"42.37", parsed value: 42
For input"1234567890123345", parsed value: -2147483648
For input"2023C++31", parsed value: 2023

Figure 20.4: Output from LookAtFromChars

Speaking of features, it seems probable that these functions will get a span variation in a future release, but not for C++23.

Recipe 20.7: constexpr std::bitset

Problem: We want a safer way of checking our std::bitset result at compile-time.

Solution: C++ 23 now has the member functions being

Not that we had to focus on std::bitset to stress out point, but having so many more functions being constexpr allows much more safety in terms of being able to check (and fail) early.

```
{



{

<< << << __FUNCTION__ <<

b1{ "00101010" };

==

}

};
```

Being able to have your compiler, or even your IDE, tell you that something is failing, means that you can diagnose the problem right away, making it fewer (or even no) build cycles to diagnose a problem.

Recipe 20.8: std::to_underlying

Problem: We want a safer way of checking if our enum value is the appropriate integer value.

Solution: C++ 23 has the new std::to_underlying() function.

Most developers do not stray from the norm and use different base types for the enum values. When the need requires to use different types, it would be good to check if the enum has the end correct value. If you were using library code, to validate this might force you to make a lot of code with the various checks. Thankfully in C++23, there is a new function that does just that. It is also so you can do some of your checks at compile-time.

An example of the usage is as follows:

{

{

<< << << __FUNCTION__ <<

: char {

Breakfast,

Lunch,

Dinner,

};

: long {

SoupSpoon,

DessertSpoon,

ButterKnife,

OysterFork,

SaladFork,

DinnerFork,

DinnerKnife,

DessertFork,

```
  };


: unsigned {

AngelHair,

EggNoodle,

  Fettuccine,

  Fusilli,

  Lasagna,

  Linguine,

  Macaroni,

  Manicotti,

  Orzo,

  Penne,

  Rigatoni,
```

```
    Rotelle,

    Rotini,

    Spaghetti,

    };


}

};
```

Note the last enum does not specify class or struct, and so you can (dangerously) specify a value without specifying the enum scope.

Recipe 20.9: Direct initialization

Problem: We want a safer way of doing initialization.

Solution: C++ 20 has made a number of small changes relating to initialization.

In a previous chapter, we have already talked about designated initializers, where you specify which variable you want to initialize. While you agree that makes the code safer, that is not the focus. It is relating to initializing aggregates, which is what we want to talk about. C++20 cleaned up a number of small issues relating to initialization. (See references)

The first was more of an inconsistency than an attempt to make your code safer. In the following example, we have removed the constructor of our Thing class. That means that you cannot just create an object. In C++17, you could create it with an empty initializer. That ability, albeit a breaking change, seemed worthwhile to clean up in C++20.

Another inconsistency dealt with parentheses versus braces, and so now you should, in most situations, declare an object with either. One of the problems was for asserts, or other macros. You could instantiate an object with parenthesis within a macro, but the compiler did not understand when you tried to instantiate an object with braces. There were also problems when trying to do perfect forwarding in templates, and so emplace and make_unique did not work for aggregates. Since C++20,

aggregate initialization can be with either, and will work in the same manner, except that:

Parenthesis () does not

Braces { not allow narrowing conversions.

But the suggestion is, that if you know your code is correct, if one fails then try the other approach.

Another initialization feature relates to range-based loops. In our example, when you call it may be library code that you do not know how it works, and when you call it might be trying to loop on a temporary variable, which could cause undefined behaviour. In C++20, they defined an initializer that you could have before your loop. This would mean your variable stays in scope for that loop, and you do not have to worry about it.

```
{

Thing() =

};

{

int x;
```

```
int y;

};

{};

{

// other code

{ }

};

Menu { return Menu{}; }

{ }

{

{

<< << << __FUNCTION__ <<

// Thing t1; // cannot create because constructor is deleted
```

```
// Thing t2 {}; // after C++17, can no longer instantiate with aggregate initialization



OtherThing 67 };

int

int };

//for (auto&& special : GetMenu().GetSpecials()) { // may not be safe

for menu = GetMenu(); auto && special : menu.GetSpecials()) {

  }

auto [x, y] = GetOtherThing();

[x2, y2] = GetOtherThing(); // now works in C++20

[x3, y3] = GetOtherThing(); // now works in C++20

auto fn = [x] { return x > }; // fixed problem that this did not work in C++17

}
```

```
};
```

One last change related to initialization relates to structured bindings. The last code block gets the OtherThing object, but instead of looking at that object, it binds to a structure where it decomposes the return value. Before C++20, you could not do this with static or thread_local values, nor could you use the values inside a lambda. While this might be labeled as fixing some bugs or shortcomings, the abilities now help with your code, and thus make the code safer.

Conclusion

At the beginning of the chapter, we mentioned that the goal should be to make your code safer. While in the end, you are responsible for making your code safer, all these tools should make your task a lot easier.

Points to remember

Do not forgot all the other features from the previous chapters, as to how they can make your code safer.

Using the right types, and validating that you are using the right type, goes a long way towards code safety.

Use explicit wherever you think it is appropriate since that will stop you from unknowingly creating objects.

References

YouTube. (n.d.) ConditionalExplicit. Retrieved from https://www.youtube.com/watch?v=CMmkoy24XhU

YouTube. (n.d.). Retrieved from https://youtu.be/E_GoY-WGgmA

Wikipedia. (n.d.). BrainFloat. Retrieved from https://en.wikipedia.org/wiki/Bfloat16_floating-point_format

YouTube. (n.d.). AvoidPunning. Retrieved from https://www.youtube.com/watch?v=pbkQG09grFw

(n.d.) P1132. Retrieved from https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p1132r8.html

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 21

Making Your Code Faster and Easier to Debug

Introduction

This chapter touches upon a couple of different aspects. The first deals with speed. There are a few things to mention that will make your code go faster. We will mention some new keywords to help the compiler work better. Hopefully if the compiler can optimize your code a bit better, then that means faster code as well. We will also talk about some nice new features that should make it easier to debug your code, and make the debugging process faster.

Structure

In this chapter, we will cover the following topics:

Recipe 21.1: [[likely]] and [[unlikely]]

Recipe 21.2: [[assume()]]

Recipe 21.3: std::unreachable

Recipe 21.4: string::substr is faster

Recipe 21.5: string::resize_and_overwrite

Recipe 21.6: std::endl vs "\n"

Recipe 21.7: Comparing integrals

Recipe 21.8: PMR containers

Recipe 21.9: Synchronized stream output

Recipe 21.10: std::source_location

Recipe 21.11: std:stacktrace

The features being discussed were mostly complete, with a few exceptions:

}

Objectives

By the end of this chapter, the reader will understand some new ways to speed up their code, and new ways to make their debugging process go faster. You have learned from previous chapters new classes and keywords that will improve your productivity. While some of these topics are not as large, they too, will help you. One can be productive in different ways; whether just based on how fast you can create code, but also how on efficiently you are able to diagnose issues that arise from your code. You will learn some new types that will help you get your code up to production level quality faster.

Recipe 21.1: [[likely]] and [[unlikely]]

We want to help the compiler optimize for paths of execution.

C++ 20 has new attributes named likely and

With speed being as important as ever, it is good to do whatever you can in terms of improving performance. Since compilers are so smart nowadays in optimizing code, it is a nice when we can help it along, and tell what code paths are most likely to be executed. That is where these two new attributes come in. They are a little extra help telling the compiler what you expect should happen, and let the compiler see how it can best optimize that part of the code. Unlike the attribute or the std::unreachable() function that we are about to talk about in the next couple of topics, the attributes likely and unlikely are only a guide for compiler optimization.

First let us list our headers for this chapter:

```
#include

#include

#include

#include
```

```
#include

#include

#include

#include

#include

#include

#include

#include

#include


#include

#include

#include

#include
```

```
#include

#if >= 202011L

#include

#endif

#include

#include
```

If you look at our code sample:

```
{



{}



{

<< << << __FUNCTION__ <<

  0 };
```

```
for = i < i++) {

switch

  {

[[fallthrough]];

[[fallthrough]];

  [[likely]]

<< "In case 2, for: " << i <<




<< "In case 3, for: " << i <<


  [[unlikely]]

<< "In case 4, for: " << i <<


// maybe_unused cannot be used in case blocks
```

```
//[[maybe_unused]] case 99:

// std::cout << «In case 99, for: « << i <<

// break;

  }

  }

}

};
```

We can see in the following output in Figure that the unlikely attribute does not affect what the output says. Adding the attributes may affect how fast the code is executed though.

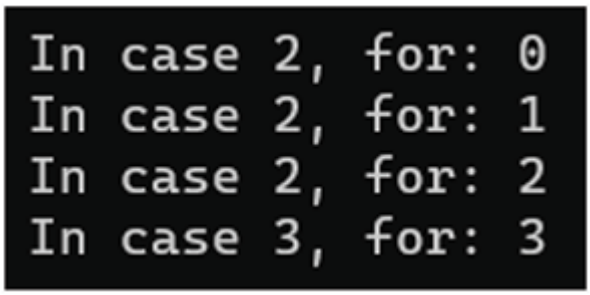

Figure 21.1: Output for LookAtLikely

We will point out, that although to some seeing these attributes in the code may be a bit of a distraction, for other developers, it also could be a learning aid. Instead of a comment saying // it will probably run this saying it explicitly with a likely attribute could also help other developers understand what is happening in the code.

On another note, we sprinkled a couple of other attributes in our code example, for reference. We are showing the C++17 attribute maybe_unused at the beginning of a function, and in front of a variable. We cannot place maybe_unused in a case block. We included it to show that the maybe_unused attribute is not about code optimization, but for compiler warnings. If you are trying to reduce the number of warnings you have in your build (which you should be doing), then you can place this where a variable or function is not used due to #ifdef or some other macro logic. If you do find unused code still in the codebase for no reason, then you should consider removing it. This usage should be for entities that are sometimes included in a build, but perhaps not always. If the compiler sees something in a build that is not used, then it might issue a warning. If this is a false positive, then the maybe_ususued attribute will suppress it.

The other attribute we have in the code is named When C++ was designed, for some reason, the default action was very oddly planned. It executes all the case blocks below it, until it sees a break statement. In most cases, we only want the one block, but there is the odd time when we want to execute more than one case block of code. Compilers are smart enough to know that not having a break statement looks suspicious, and issues a warning. To suppress that warning, you can add the fallthrough attribute to tell it that is your intent.

Given all of what we have said, these attributes could improve performance, but more likely an up-to-date compiler, with a good level of optimization, will probably give you better performance results. Perhaps these attributes are too new, and eventually the compiler will reap more benefit from them. For now, if nothing else, some would say it helps self-document the code.

Recipe 21.2: [[assume()]]

We want our compiler to be a bit more optimized by telling it extra information.

C++23 has a new attribute named that you can give hints to the compiler on what to think.

Another aid for the compiler is the new assume attribute. This should not affect the code execution, and it does state the expression given should be a null statement, meaning it should not be part of the code's logic. The expression should evaluate to true, otherwise will be undefined behaviour. These statements should just exist to help the compiler optimize how to interpret the code. The given expression is meaning unless it includes: or then it is not guaranteed to be evaluated. (See references)

{

{ }

{ }

{

```
<< << << __FUNCTION__ <<

= ReturnsHour();

  <

// [[assume(CriticalThisGetsRun())]]; // wrong: do not put essential
methods in here

  ==

/* will now be undefined behaviour, since compile will assume hr has
wrong value */



}

};
```

If you look at the code example, our assume statements are simple statements to help the compiler know the current state. You should not put actual functions that need to be run in assume statements. You should also be certain the assumption is correct. Otherwise, the compiler may make wrong guesses and the execution will be undefined. For some compilers, if assume fails, execution may even be aborted. If you really do not want the code to execute if the assumption Is false, then you explicitly should use an assertion (which will abort the program if it fails), or have an if statement that throws an exception (which should be caught/handled elsewhere).

An extra little note, there are not any compile-time feature checks for
None of the compilers we are looking at had this feature complete, and we received compiler warnings, but at least the code built and ran.

Recipe 21.3: std::unreachable

We want a mechanism to denote that code path should not be entered.

C++23 has a new function named

We have talked about the [[unlikely]] attribute, which is for the compiler to optimize code but will not affect the execution result. We then talked about the [[assume]] attribute, which tells the compiler the current state of execution, and if that is false, then the behavior is undefined. This topic deals with the new unreachable function, which if we get to that point, then we know the execution is in a bad state and say it is undefined behaviour. It is compiler-specific, but normally that means to abort the program. The compiler may make optimization assumptions for a release build and remove that code. Thus, you should find out for your compiler the particular behavior and whether you want to use or or throw an exception. If you look at the following code example:

```
{



{

switch

  {
```

```
[[fallthrough]];

[[fallthrough]];


<< "In case 2, for: " << i <<



<< "In case 3, for: " << i <<




#if >= 202202L


#else

<< "Feature not yet implemented for this compiler." <<

#endif

  }

}
```

```
{

<< << << __FUNCTION__ <<

for = i < i++) {

UnreachableHelper(i);

  }

//

//UnreachableHelper(10); // now has undefined behaviour

}

};
```

You can see that we reused code similar to our previous example. If we get to someplace in the code where the unreachable would be executed, it may be already optimized out or could cause the program to abort, depending on your compiler and build.

Recipe 21.4: string::substr is faster

We want a faster string::substr() function.

C++23 has a performance improvement.

A small change, but every little bit helps. If you look at the following code example, the change is that previously the GrabSomeText() function would return a temporary variable, and the substr() function would create a new string and then copy over the content. The performance change is that now GrabSomeText() returns a normal string, and the substr() function does not make its own copy.

```
{


{


}


{

<< << << __FUNCTION__ <<
```

```
auto s =

<< "Text= " << s <<

}

};
```

Recipe 21.5: string::resize_and_overwrite

We want a faster way to resize strings.

The C++23 release gives us

The std::string is extremely handy, but not the most performant. It is especially so, when developers get lazy and just keep appending to the existing value. This often makes the string continuously do incremental internal resizing. If you knew you were going to add a lot of content to the string, it makes more sense to increase the size in one big chunk. That is what the resize() function does, and in C++20, they made that

In C++23, they added a new function called This can also resize to give you extra memory (or less if that is your goal), but the main idea is that you have control over how you set the string contents. This could mean that you perform move operations and not copies, thus saving a lot of time. Look at the following sample code:

```
{



input, size_t count) {

result;
```

```
input.size() };

step * count };

result.resize_and_overwrite(total, buf, size_t n) {

for = i < count; i++) {

+ * step), input.data(), step);

  }

return total;



  });

return result;

}



{

<< << << __FUNCTION__ <<

result = is
```

```
<< result <<

}

};
```

You can see in our case, we are performing a memcpy() of our input to the character address. Moreover, unlike calling it does not initialize the memory, since it knows it will be overwritten. This will be handy when you already have a value that you want to update your string with. Now you do not have to do two steps of resizing and then updating the contents.

Recipe 21.6: PMR containers

We want a fast way of pre-allocating storage.

C++17 offered up some functionality that is still worth taking a look.

To segue from talking about string allocations, since we are talking about performance, it seemed appropriate to talk briefly about Polymorphic Memory Resource and custom allocators. While not a brand-new feature, this C++17 namespace does not seem to get enough attention. It could be extremely useful for those who need extremely fast access to something within a chunk of memory that they know the size of. The performance comes from the memory being pre-allocated, and in a contiguous block, so there is less fragmentation.

Some of the std::pmr classes are a lot more complex than what we will look at here. You can assign your own allocate and deallocate functions if you do not want to use the default new and You can also provide an alignment size, to help improve memory access.

Our example is using a class named but there are others:

Fast, but not thread-safe allocation into a buffer.

Pools of similar sized memory blocks, but not thread-safe.

Pools of similar sized memory blocks, and thread-safe.

Allocates using the new operator.

The monotonic_buffer_resource class is the fastest, but the downside is that it does not deallocate. It only deallocates when you are completely done with the resource. This could be fine, because we are saying usage for this sort of thing is specialized where we want to allocate a chunk of memory and keep it around for a while.

```
{



{

<< << << __FUNCTION__ <<



pmrVector1{ { 6 }, &resource };

pmrVector2{ { 12 }, &resource };

auto DumpList = name, v) {

<< <<
```

```
for : v) {

<< i << ",

  }

<<

  };

pmrVector1);

pmrVector2);

}


};
```

The output is not exciting, it is just to show that the vectors are looking at different areas of pre-allocated memory. Refer to Figure

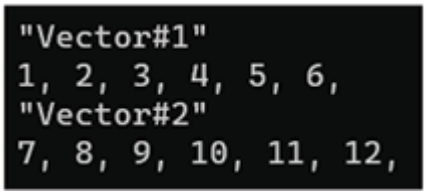

Figure 21.2: Output from LookAtPMR

While there are lots of options on how to do things, the downside is that you need to do some research to decide what options are best for your situation.

The interesting thing about PMR functionality is that, if you say you have not really heard of it, then you would be shocked to know that you have probably scrolled through it being mentioned many times. If you go back and look at one of the standard containers on you will see if the case is mentioned for the sibling class in the pmr namespace. (See references) There is also a pmr::basic_string as well that can be utilized. If you are building a container full of strings, then both could use that pre-allocated memory, that would not have to be slow and do slow incremental size increases. You would be using the same buffer for the container, as well as the strings, and it would stay there until you knew you were finished.

Recipe 21.7: Comparing integrals

We want a fast way of comparing.

C++20 has several new functions for comparing.

This is one of those scary C++ things, where if you do not realize what can happen, then it is easy to fall prey to a bug. Although it does not sound dangerous to compare different types of integers, comparing signed and unsigned values are not safe. It is rather surprising that it took until C++20, for there to be official functions to do this properly. If you look at the following code sample:

```
{



{

<< << << __FUNCTION__ <<

int a =

b =

<<
```

```
#if >= 202002L

<< a << " == " << b << " says " << (a == b) << ", but is actually " << b)
<<

<< a << " != " << b << " says " << (a != b) << ", but is actually " << b) <<

<< a << " < " << b << " says " << (a < b) << ", but is actually " << b) <<


<< a << " <= " << b << " says " << (a <= b) << ", but is actually " << b)
<<

<< a << " > " << b << " says " << (a > b) << ", but is actually " << b) <<

<< a << " >= " << b << " says " << (a >= b) << ", but is actually " << b)
<<

#else

<< "Feature not yet implemented for this compiler." <<

#endif

}

};
```

Simple comparison between a signed and an unsigned does not sound that bad, but if there was a negative value, then most compilers take the wrong approach and treat it is a large positive value. This means comparison operators give the wrong result, as you can see in Figure

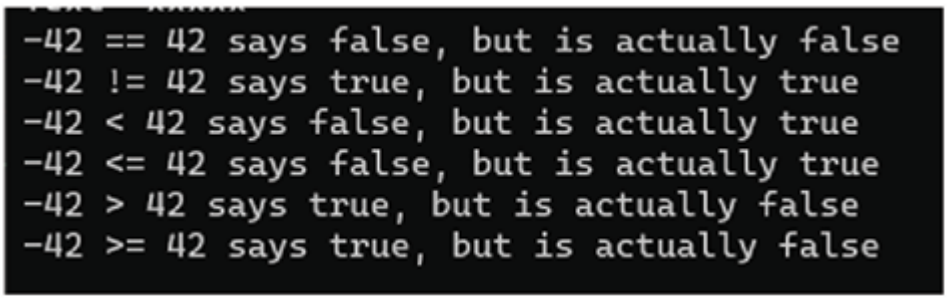

Figure 21.3: Output from LookAtComparingIntegrals

Ideally, if the values are for the same purpose, then they should all be of the same type. If they cannot be the same type, then these new functions now provide a standard way of accurately comparing them.

Recipe 21.8: std::endl vs "\n"

We want a fast way of outputting our text.

Be careful about how you end your output lines.

We are actually not going to talk about a new feature, but just a reminder about something, and an acknowledgement of what we have been doing in the book.

In most of our coding examples, we have been outputting some text to and then sending a While the output result looks the same, there is a performance difference. The std::endl will also flush the output, which could take a considerable amount of time. If you look at the following sample:

{

{

<< << << __FUNCTION__ <<

= {

=

```
for = i < i++) {

outfile << "On line " << i;

if {

outfile <<

    }

else {

outfile <<

    }


  }

outfile.close();

=

delta = timerEnd - timerStart;
```

```
return delta;

  };

durationWithEndl =

durationWithN =

<< "Took " << << durationWithEndl.count() << " ticks using endl"

  << ", and\ntook " << << durationWithN.count() << " ticks with \\n" <<

}

};
```

We created a lambda that writes a line to a file, and after each line, we send a new-line string, or send Although the times are not shockingly poor, there is a considerable difference. Refer to Figure

Figure 21.4: Output from LookAtEndl

We have mostly std::endl in our code examples because they seem clearer on what we want to do. There are probably many things we would do differently if the code samples were needed to be of production quality. For now, we will say that the sample code's highest priority was to explain what a new feature was, and not to be the most performant possible. Since these are little snippets of text sent to the standard output, in most cases we just use

Recipe 21.9: Synchronized stream output

We need a way of having our output text organized.

C++20 has a new class for synchronizing output.

Still on the topic of output, but we will shift from making your code faster to making debugging faster. Often it is a big task to look at a log file to understand what is going on. If you have multiple threads. The log entries are all jumbled together.

This technique does not have to be just with log files. If you have multiple threads that write to the same file, then you will greatly appreciate This new class wraps around an existing stream, to enjoy your write calls to not get jumbled together. (See references)

If you look at our sample code, we are doing the same thing twice. In our restaurant, we have a few different customers come in asking for multiple orders of the same dish:

```
{



dish, int orderCount, int msCookTime) {

<< "Told to make " << orderCount << " orders of " << dish <<
```

```
for orderNumber = orderNumber <= orderCount; ++orderNumber) {


<< dish << " # " << orderNumber << " is

  }

<< "Done making " << dish <<

}


orderRequests) {

= orderRequests.size();

=

for n = n < numRequests; n++) {



  }

for n = n < numRequests; n++) {

messyThreads[n].join();
```

```
  }

<< "All messy orders are ready." << <<

}


{

<< << << __FUNCTION__ <<

orderRequests{

{

{ "fried 350 },

{ "spinach 100 }, };

LookAtMessyOutput(orderRequests);

}

};
```

We create a new thread for each customer, and log when we start and finish each dish we make for the customers. The resulting output is rather messy, as shown in Figure

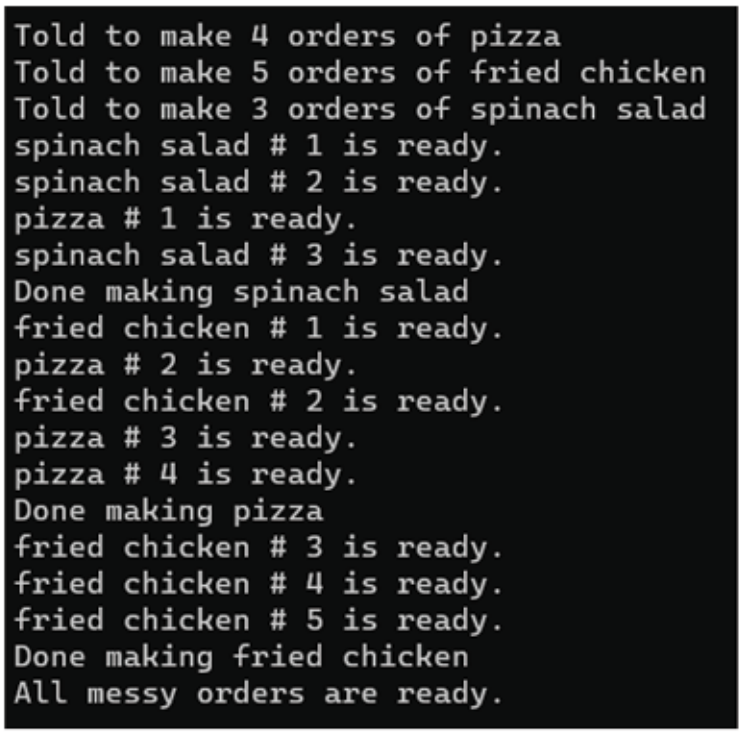
Told to make 4 orders of pizza
Told to make 5 orders of fried chicken
Told to make 3 orders of spinach salad
spinach salad # 1 is ready.
spinach salad # 2 is ready.
pizza # 1 is ready.
spinach salad # 3 is ready.
Done making spinach salad
fried chicken # 1 is ready.
pizza # 2 is ready.
fried chicken # 2 is ready.
pizza # 3 is ready.
pizza # 4 is ready.
Done making pizza
fried chicken # 3 is ready.
fried chicken # 4 is ready.
fried chicken # 5 is ready.
Done making fried chicken
All messy orders are ready.

Figure 21.5: Output from LookAtMessyOutput

Although it might be accurate time-wise, it is confusing to understand the flow.

If we try the same thing, but wrap the std::out so it can be synchronized, the code does not look any different:

{

dish, int orderCount, int msCookTime) {

auto started =

```
  syncOutput << "Told to make " << orderCount << " orders of " << dish << "
@ " << started <<

for orderNumber = orderNumber <= orderCount; ++orderNumber) {



syncOutput << dish << " # " << orderNumber << " is

  }

auto finished =

  syncOutput << "Done making " << dish << " @ " << finished <<

}

orderRequests) {

= orderRequests.size();

auto threads =

for n = n < numRequests; n++) {
```

```
  }

for n = n < numRequests; n++) {



  }


<< "All tidy orders are ready." << <<

}


{


<< << << __FUNCTION__ <<

orderRequests{

{

{ "fried 350 },

{ "spinach 100 }, };

LookAtCleanOutput(orderRequests);
```

```
}

};
```

However, the output is much cleaner, as can be seen in Figure

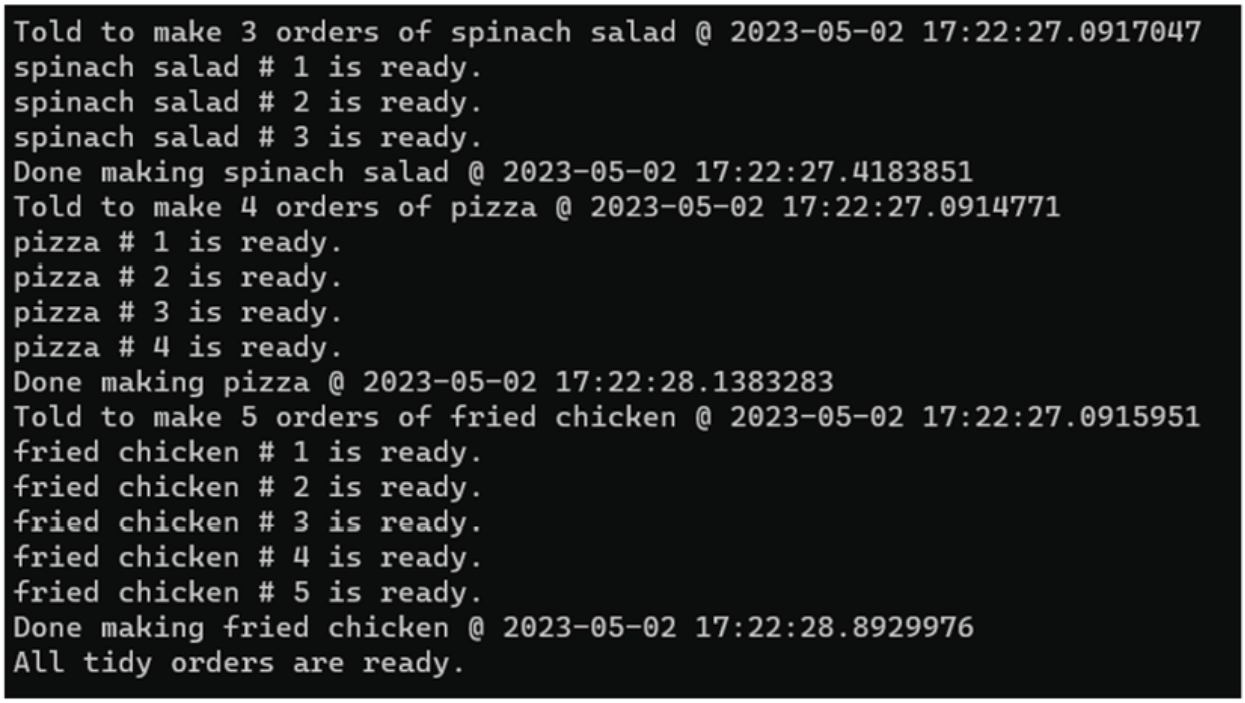
Told to make 3 orders of spinach salad @ 2023-05-02 17:22:27.0917047
spinach salad # 1 is ready.
spinach salad # 2 is ready.
spinach salad # 3 is ready.
Done making spinach salad @ 2023-05-02 17:22:27.4183851
Told to make 4 orders of pizza @ 2023-05-02 17:22:27.0914771
pizza # 1 is ready.
pizza # 2 is ready.
pizza # 3 is ready.
pizza # 4 is ready.
Done making pizza @ 2023-05-02 17:22:28.1383283
Told to make 5 orders of fried chicken @ 2023-05-02 17:22:27.0915951
fried chicken # 1 is ready.
fried chicken # 2 is ready.
fried chicken # 3 is ready.
fried chicken # 4 is ready.
fried chicken # 5 is ready.
Done making fried chicken @ 2023-05-02 17:22:28.8929976
All tidy orders are ready.

Figure 21.6: Output from LookAtCleanOutput

If you look at the times, we were told to start making the pizza before the salads were completed, but since the output was grouped, the salads had started first and were sent to the output together.

Certainly, in some logging cases, you want each entry to be outputted synchronously, but often it would be nice to group several entries to make it more readable.

Recipe 21.10: std::source_location

We want a better way of knowing what source code is being executed.

C++20 has a new std::source_location class.

No doubt a big part of investigating a code issue is looking at some sort of logging, and tracking back to where it says it is in the code. While most compilers provided the ability to do this, each had their own macros or custom functions. Now with C++20, things are standardized and from now on it is much better to use in code.

```
{



loc) {



}

T>

obj) {

auto loc =
```

```
<< "File=" << GetFileName(loc) << ", Fn=" << loc.function_name() << ",
Line=" << loc.line() <<

}

{

auto loc =

<< "File=" << GetFileName(loc) << ", Fn=" << loc.function_name() << ",
Line=" << loc.line() <<

42 };


DoSomethingWithT(i);

}


{

<< << << __FUNCTION__ <<

auto loc =

<< "File=" << GetFileName(loc) << ", Fn=" << loc.function_name() << ",
Line=" << loc.line() <<
```

```
DoSomething();

}

};
```

Which nicely gives us the output shown in Figure

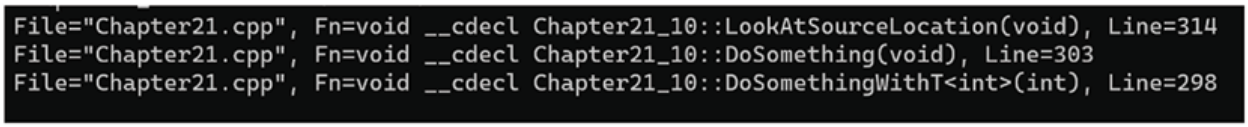
File="Chapter21.cpp", Fn=void __cdecl Chapter21_10::LookAtSourceLocation(void), Line=314
File="Chapter21.cpp", Fn=void __cdecl Chapter21_10::DoSomething(void), Line=303
File="Chapter21.cpp", Fn=void __cdecl Chapter21_10::DoSomethingWithT<int>(int), Line=298

Figure 21.7: Output from LookAtSourceLocation

Simply calling the static std::source_location::current() will provide the (full path) file of the source, the function and the line number.

Recipe 21.11: std:stacktrace

We want a nice way of knowing the code-path of how we got to some specific code.

C++23 has a new std::stacktrace class.

As developers, when we are debugging, we have learned to look at the stack trace to understand what is going on in the code. At last, in C++23, we can look at the stack-trace in code. We might do that to dissect what went wrong somewhere, we might log in, or we might analyze it to understand the user's patterns. We can get the current stack trace by calling the static current() function. This will give us a container of std::stacktrace_entry objects.

The current() function has a few variations, and aside from the default, probably the next most used one would be to say how many entries we want to skip over. In an earlier chapter, we mentioned that there were custom formatters set up to handle specific types. Thankfully, they built in formatters for std::stacktrace and so it should, by default, give you something presentable when you print it out. We can iterate the container of entries, although C++23 has provided a default formatter, sending it to a stream is readable. Each entry contains a description, which is normally the function name, the file name, and the line number. Curiously, they did not reuse the std::source_location class. If we look at the following code example, we created our own exception just to illustrate a point. In constructor, we capture what is going on with std::stacktrace::current(1). The value 1 indicates that we want to skip over the first stack-trace entry. Since we are gathering the stack-trace in the exception's constructor, we do not need to keep that information.

```
: {


message) : message } {

#if >= 202011L

  Trace =

#endif

}

#if >= 202011L

{ return Trace; }


Trace;

#endif

};

{
```

```
T>

obj) {


}

{



}

{

  LookAtStacktrace3();

}


{


<< << << __FUNCTION__ <<

#if >= 202011L
```

```
try {

  LookAtStacktrace2();

  }

catch ex) {

auto problem = + ": " + ex.what();

auto trace = ex.GetTrace();

<< "\n" << problem << << trace <<

=

<< "\nfirst stacktrace_entry= " << firstEntry <<

<< "entry details: description=" << firstEntry.description()

    << ", file=" << << ", line=" << firstEntry.source_line() <<

}

#else

<< "Feature not yet implemented for this compiler." <<
```

```
#endif

}

};
```

We created a few functions that we call before we create our exception. Our output shows:

```
class MyException: oops!
0> D:\Dev\_To put into repo\BookSamples\BookSamples\Chapter21.cpp(265):
BookSamples!LookAtStacktrace4T<int>+0x41
1> D:\Dev\_To put into repo\BookSamples\BookSamples\Chapter21.cpp(269):
BookSamples!LookAtStacktrace3+0x21
2> D:\Dev\_To put into repo\BookSamples\BookSamples\Chapter21.cpp(272):
BookSamples!LookAtStacktrace2+0x1C
3> D:\Dev\_To put into repo\BookSamples\BookSamples\Chapter21.cpp(275):
BookSamples!LookAtStacktrace+0x31
4> D:\Dev\_To put into repo\BookSamples\BookSamples\Chapter21.cpp(300):
BookSamples!Chapter21::LookAtChapterTopics+0x1C
5> D:\Dev\_To put into repo\BookSamples\BookSamples\BookSamples.cpp(63):
 BookSamples!main+0x1C
6> D:\a\_work\1\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl(7
9): BookSamples!invoke_main+0x39
7> D:\a\_work\1\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl(2
88): BookSamples!__scrt_common_main_seh+0x12E
8> D:\a\_work\1\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl(3
31): BookSamples!__scrt_common_main+0xE
9> D:\a\_work\1\s\src\vctools\crt\vcstartup\src\startup\exe_main.cpp(17)
: BookSamples!mainCRTStartup+0xE
10> KERNEL32!BaseThreadInitThunk+0x1D
11> ntdll!RtlUserThreadStart+0x28

first stacktrace_entry= D:\Dev\_To put into repo\BookSamples\BookSamples
\Chapter21.cpp(265): BookSamples!LookAtStacktrace4T<int>+0x41
entry details: description=BookSamples!LookAtStacktrace4T<int>+0x41, fil
e=D:\Dev\_To put into repo\BookSamples\BookSamples\Chapter21.cpp, line=2
65
```

Figure 21.8: Output for LookAtStacktrace

The first entry is now which is the templated function where we create and throw our exception. As expected, we see the calls of the other functions, with their file name, function name and line number. We then grab the first stacktrace_entry from the container, and when we print it, unsurprisingly it looks the same as from when we printed out the full stack-trace. We then printed out the value from the entry, just to see that they are the same. Adding this information to your logging will increase their size, but have this extra information will hopefully increase your productivity.

Conclusion

At the beginning of the chapter, we said we will talk about having things faster. We can mention again, that all the const-related keywords should make development time faster by finding issues at compile-time, rather than at run-time. We can also mention that with all the new range and view functionality, that you will be able to do more, with less code, a lot faster than before. There are many other points we could mention, but we will just say that recent C++ releases have made things a lot better for developers.

In this chapter, we learned about some things that will make it better for compilers. There are new attributes you can use to help the compiler optimize the code for you, and make it run faster. There are also new functions that will make working with strings faster. Having synchronized streams, and a stack-trace should make debugging a lot easier, so the time it takes to resolve issues will be faster.

Points to remember

The compiler is our friend. Treat it nicely by providing the and assume attributes, and it will reward you with more optimized code.

Make your life easier and debugging odd problems faster, and when trying to compare integers of different types, use the new comparison functions.

The PMR functionality is not for everybody, and everywhere, but for those specific situations it can really make your code faster.

Similar advice to use synchronized streams. For when you do not want different threads confusing your output, the new mechanism will make the debugging process better.

Being able to log the source location and stack-trace, will certainly make resolving issues go faster.

References

CPPReference. (n.d.). Potentially. Retrieved from https://en.cppreference.com/w/cpp/language/expressions#Potentially-evaluated_expressions

ModernesCPP. (n.d.). IntegralComparison. Retrieved from https://www.modernescpp.com/index.php/safe-comparisons-of-integrals-with-c-20

ModernesCPP. (n.d.). SyncStream. Retrieved from https://www.modernescpp.com/index.php/synchronized-outputstreams

YouTube. (n.d.). PMR. Retrieved from https://www.youtube.com/watch?v=v3dz-AKOVL8

CppReference. (n.d.). PmrList. Retrieved from https://en.cppreference.com/w/cpp/container/list

Index

Symbols

header

pi and e

#warning command 542

A

aggregate

initializing

assignable_from

suggestion 36

assume attribute 560

atomic 7

auto

transforming 511

B

basic range access functions

bidirectional_iterator

binary_find() free function 441

Binary Module Interface (BMI) 260

binary search functions

bit initialization 528

bit manipulation

C

C++ Weekly 141

char8_t type 512

chunk() adaptor

using

Class Template Argument Deduction (CTAD) 505

improvements

clock_cast() function 471

clocks

file_clock 471

gps_clock 471

high_resolution_clock 471

local_t 471

steady_clock 471

system_clock 471

tai_clock 471

utc_clock 471

common_reference_with

common_with

comparison functions

Compiled Module Interface (CMI) 260

Compiler Explorer 141

compound requirement

concepts 2

first concept, creating 4

suggestion 6

using 5

conditionally explicit constructor

conjunction

creating 7

suggestion 8

consteval

constexpr

enhancements

for transient allocation

hidden problems 174

constexpr if

constexpr keyword 139

constexpr std::bitset 549

constinit

constraint

creating 6

constructible_from

containers 427

contains() function

contiguous_iterator

convertible_to

copyable

copy_constructible

coroutines

components

restrictions

state and handle

count() function

count_if() function

Curiously Recurring Template Pattern (CRTP) 384

custom range factory

creating 417

cyclical modules

D

dates

day 465

month 465

month_day 465

month_day_last 466

month_weekday 466

month_weekday_last 466

new features

weekday 465

weekday_indexed 465

weekday_last 465

year 465

year_month 466

year_month_day 466

year_month_day_last 466

year_month_weekday_last 466

default arguments

adding, for pair's forwarding constructors 443

default_initializable

derived_from

destructible

direct initialization

drop() adaptor

using

E

enable_view flag 381

enum

using

equality_comparable

equality_comparable_with 60

equivalence_relation 134

F

factory overview

fallthrough attribute 559

filter() adaptor

using

final_suspend() function 241

fixed width floating-point types 535

floating_point 33

suggestion 34

fold functions

for_each functions

forward_iterator

function

constraining 9

function templates 12

G

generators

get_tzdb_list() function 474

H

heap functions

I

if consteval

incrementable 93

indirect_binary_predicate 136

indirect_equivalence_relation 136

indirectly_comparable 109

indirectly_copyable 109

indirectly_copy_storable 109

indirectly_movable 108

indirectly_movable_storable 108

indirectly_readable 92

indirectly_regular_unary_invocable 136

indirectly_swappable 109

indirectly_unary_invocable 136

indirectly_writable 92

indirect_strict_weak_order 137

IngredientList variable 16

init-statement

extending, for alias-declaration 529

input_iterator 93

input_or_output_iterator

integrals 31

comparing 567

invocable

invoke_r 526

IsChickenCooked() function 91

is_clock() function 470

iterators pair constructors

for stack and queue 442

J

join() adaptor

working with

K

keys() adaptor

working with

L

lambda parameter packs

lazy_split() 395

likely attribute

literal class types

in non-type template parameters

LookAtOptional_Recap() function 516

M

MakeSomething() function 517

maybe_unused attribute 559

mergeable 110

minimum and maximum functions

module

compiler, using 260

layout

partitions 275

using

movable

move_constructible

multidimensional subscript operator

N

named universal character escapes 514

nested requirements 11

O

One Definition Rule (ODR) 253

operator() function 305

output_iterator 93

owning_view 295

P

permutable 110

permutation functions

Polymorphic Memory Resource (PMR) containers

predicate 134

pre-processing directives

elifdef 525

elifndef 525

PrintRange() function 358

pseudo-clock 471

push_range() function

for queue, stack and priority_queue 444

R

random_access_iterator

range adaptor

using, with composition

range concepts

range primitives 287

ranges

enhancements 461

formatting

swapping

ranges::binary_search() function 343

ranges::clamp() 334

ranges::common_view() 406

ranges::construct_at() 361

ranges::copy() 365

ranges::copy_backward() 362

ranges::copy_if() 367

ranges::dangling

ranges::empty_view<> 384

ranges::ends_with() 325

ranges::equal_range() 344

ranges::fill() 366

ranges::fill_n() 366

ranges::generate() 376

ranges::generate_n() 376

ranges::greater() function 338

ranges::inplace_merge() 347

ranges::iota_view factory

ranges::is_heap() 360

ranges::is_sorted() 335

ranges::is_sorted_until() function 341

ranges::join_view() adaptor 392

ranges::lower_bound() 343

ranges::make_heap() 360

ranges::move_backward() function 365

ranges::move() function 365

ranges::next_permutation() 348

ranges:: nth_element() 341

ranges::owning_view() 406

ranges::partial_sort() 340

ranges::partial_sort_copy()

ranges::partition_copy() 342

ranges::pop_heap() 360

ranges::prev_permutation() 348

ranges::push_heap() 360

ranges::ref_view() 406

ranges::remove() 366

ranges::remove_copy_if() 367

ranges::remove_if() 366

ranges::remove_if() algorithm 390

ranges::replace() 370

ranges::replace_copy() 370

ranges::replace_if() 370

ranges::reverse() 372

ranges::reverse_copy() 372

ranges::rotate() 369

ranges::rotate_copy() 369

ranges::sample() 373

ranges::set_difference() 345

ranges::set_intersection() 345

ranges::set_union() 345

ranges::shift_left() 367

ranges::shift_right() 367

ranges::shuffle() 373

ranges::shuffle_copy() 373

ranges::single_view<> 384

ranges::sort() 373

ranges::sort_heap() 360

ranges::stable_partition() 342

ranges::stable_sort() function 338

ranges::starts_with() 325

ranges::swap() function 372

ranges::swap_ranges() 371

ranges::symmetric_difference() 346

ranges::to utility 425

ranges::transform() 390

ranges::uninitialized_copy() 360

ranges::uninitialized_default_construct() 361

ranges::uninitialized_default_construct_n() 361

ranges::uninitialized_destroy() 361

ranges::uninitialized_destroy_at() 361

ranges::uninitialized_destroy_n() 361

ranges::uninitialized_fill() 361

ranges::uninitialized_fill_n() 361

ranges::uninitialized_move() 361

ranges::uninitialized_move_n() 361

ranges::uninitialized_value_construct() 361

ranges::uninitialized_value_construct_n() 361

ranges::unique() 373

ranges::unique_copy() 373

ranges::upper_bound() 343

ranges::view_interface<> 384

ranges::views::lazy_split 395

reachability

regular

regular_invocable 132

relation 134

requires clause

creating 9

requires expression

creating 10

multiple requires expressions 10

Resource Acquisition Is Initialization (RAII) 113

reverse() 405

Rule of Five states 113

Rule of Six 113

Rule of Three 113

Rule of Three states 112

S

same_as

search functionality

semiregular

sentinel_for

sequence functions

modifying

set functions

signed_integral 31

simple requirement 10

sized_sentinel_for 91

slide() adaptor

using

SomeNoceptFunction 14

sorting and partitioning functions

span 283

special adaptors

specializations

split() adaptor

working with

standard modules

importing 274

starts_with and ends_with

for strings

std::atomic

std::atomic

std::atomic::notify

std::atomic::wait

std::atomic_ref

std::back_inserter() function 346

std::barrier

std::binary_semaphore

std::bit_ceil() 483

std::bit_width() 484

std::bloat16_t 535

std::counted_iterator

std::counting_semaphore

std::endl

versus, "\n"

std::erase() function

std::erase_if() function

std::expected

using

std::flat_map 441

std::flat_set 441

std::from_chars()

std::generate() function 416

std::inout_ptr() 543

std::is_bounded_array 436

std::is_permutation() function 350

std::is_unbounded_array 436

std::jthread

std::latch 211

std::make_shared

supporting arrays

std::mdspan

std::midpoint

std::next_permutation() function 350

std::optional 514

monadic operations for

std::out_ptr() 543

std::prev_permutation() function 350

std::print 457

std::ranges::swap() 421

std::ranges::swap_ranges function 421

std::size_t

suffix 513

std::source_location 574

std::span 432

std::stacktrace class

std::start_lifetime_as 542

std::stop_callback

std::stop_source

std::stop_token

std::swap function 421

std::swappable_with 421

std::swap_ranges function 421

std::to_array

converting, into std:array

std::to_chars()

std::to_underlying() function

std::unreachable() function 562

strict_weak_order 135

string formatting

string::resize_and_overwrite function 564

strings

enhancements 461

string::substr() function 562

subrange

Substitution Failure Is Not An Error (SFINAE) 14

swappable 37

synchronized stream output

T

take() adaptor

using

template syntax lambdas

The Rule of Zero 113

ThingWithIngredients 16

this

three_way_comparable

suggestion

with custom types

three_way_comparable_with

time

clock 470

duration 470

new features

time_point 470

timezone library

totally_ordered

suggestion 64

totally_ordered_with 64

transform() adaptor

using

type requirement 10

types of expressions 14

U

u8string type 512

uninitialized memory functions 360

unlikely attribute

unsigned_interal 31

V

values() adaptor

working with

view 282

view_interface

views::adjacent() adaptor 401

views::cartesian_product 416

views::chunk adaptor 398

views::iota() factory 412

views::repeat 416

views::zip_transform() adaptor 404

visibility

W

weakly_incrementable

Z

zip() adaptor

using 404